MASTER IN COMPUTER SCIENCE

Intrusion detection through autonomous agents

Dumont, Frédéric

Award date:
1999

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grangagnage, 21, B-5000 Namur, (Belgium)

Intrusion Detection through Autonomous Agents

Frédéric Dumont

September 1999

Supervisor: Baudouin Le Charlier

Acknowledgements

First, I would like to thank my supervisor, Baudouin Le Charlier, for his help, his advices, and his patience. Then, in no specific order, Gene Spafford and Diego Zamboni, who made my internship so enjoyable, David Corcoran, for being such a nice buddy, and all the other members of the Coast Laboratory, my family and especially my sister Catherine who took the time to correct my English.

Finally, this work could not have been done without the efforts of Donald E. Knuth, Richard M. Stallman, Larry Wall and Linus Torvalds.

Abstract

Computer security is a requirement of modern business. Intrusion Detection Systems are an essential part of that security. Several types of **IDS** must be used to ensure a maximal monitoring. Moreover, autonomous agents working together to evaluate threats is a promising idea. Therefore, it is natural to integrate **IDS** and agents into a coherent system with a single point of control. Current **AAFID**'s evolution could lead to a framework where such an integration is made easier. Part of this evolution is due to the new communication mechanism based on events.

Foreword

This dissertation is divided in two parts. The first part is an introduction to computer security and security tools. It was not meant to be an extensive description of computer security. Its purpose is to place Intrusion Detection Systems in their context.

The second part devoted to **IDS**, and mostly to **AAFID**. While I was in Purdue, I was given the opportunity to work on it. My first modifications were mostly bug fixes, but the later could not be merged into the main code without many other adaptations. Therefore, I will also explain why such a change was needed, how it was done and what else can be done.

Contents

List of Figures

1. Introduction to Computer Security

In today's Information Society, we have become dependent on the rapid flow of and easy access to data. Computers play a central role in this scheme. They can manage huge amount of information and networks make access to remote files easy and transparent. It seems we can not live without such an infrastructure.

1.1. Privacy and Integrity of Data and Computers

Information is an interesting concept. It may be duplicated at no cost, yet some people may be willing to pay to have it. Many organisations' business is to collect, process and sell information. So, although information does not look like any other goods, it seems it can behave in similar ways.

Information can be attacked in two ways. First it can be destroyed or corrupted. Secondly it can be copied. This means the original owner may find it harder to sell it. In both cases, the value of this information is lower, maybe null. It is thus natural for organisations to try to protect their data.

In addition some kind of information (private information, for instance) is subject to legal restriction regarding their use. This is common in many countries. One typical restriction is that the information cannot be made public.

Computers being the key to information, they are also at risk. Their behaviour can be changed to allow easier access to restricted data to an opponent. Or someone can try to make them stop operating to interrupt the business that depends on them. Potentially worse, one could use these computers to attack another organisation. Even if legally the owner of these computers is not liable, retaliation is a common practice. Finally, the owner could be held responsible for any kind of illegal contents found on his computers.

No one is safe in this game. Because computers must be inter-connected in order to be useful today, all are a potential target. Therefore security is mandatory.

But what is Computer Security? [26] presents it as the realization of *confidentiality, integrity and availability* in a computer system. But this definition needs to be expanded.

1.2. Security Policy

The first step to protect the system is to use appropriate tools to enforce any access restrictions needed. Identification, authentication and non-repudiation are best left to computers. Most of the time, operating systems already in use in the organisation provide such mechanisms. It is necessary to make sure that these mechanisms are complete with regard to the privacy and integrity of data.

Of course, the security level should balance with the burden imposed on users. If the users feel the need to bypass any security in order to work, then the level may be to high.

The next step is to make the whole system secure. [8] defines a secure system as one that *can be depended upon to behave as it is expected to*. So the first step is to state and formalize this behaviour. Or, as [15] says it:

> The expected behavior is formalized into the *security policy* of the computer system and governs the goals that the system must meet.

This policy allows to make assumptions about the level of security offered by the system, such as resistance to brute force attacks and to various spoffing attacks.

1.3. Vulnerabilities

Vulnerabilities are defined by [15] as functional defects that could result in violation of the security policy. I would rather first define a vulnerable system as the one that allows a violation, then try to identify the flaw that caused that violation according to the following classification:

Design flaws

> This category groups weak algorithms and softwares written with no concern for security. They cannot enforce security and should not be used for this.
>
> An important fact about security algorithms (such as encryption) is that most of them will be weak one day or another. No one would use **Enigma** coding for any serious purpose, although it proved to be very good. Modern algorithms are also being studied, and new methods to crack them are being found. This means that the assumptions we can do on software (such as " it would take billions of years to break this algorithm ") may become suddenly wrong. Yet, those assumptions are what we use to implement a security solution.
>
> Trying to fix the problem can be harder than changing the tool. This also is true for softwares initially written without security. Adding it afterwards is always a tricky task.

Implementation flaws

This category groups all defects in software (bugs) resulting in a difference between the expected functionality and the actual behaviour. The only way to fix it is to change the source code and recompile. Of course, if the source code is not available, the user must wait for an upgrade from the author. If he is unavailable, then that flaw cannot be fixed.

It is important to remember that a functional defect in some software does not imply a vulnerability. Only those that could result in a violation of the security policy regarding the confidentiality, integrity or availability of computer systems are vulnerabilities.

Use flaws

This last category groups mistakes that can be made when using security tools, such as misconfiguration or weak passwords. A simple change in the way the tools are used is enough to fix the flaw.

I mention them because too often people rely on a security tool without understanding its limit or the potential weakness that result from its interaction with other tools. No tool operates alone. The global behaviour must be observed with care.

These kinds of flaws may result in exploitable vulnerabilities. When there is no way to fix the flaw, other measures must be taken in order to restore the security level, as we will see later.

1.4. Intrusions

Intrusions are security policy violations. In short, it means that the security tools supposed to enforce this policy are not working. This definition is compatible with the one found in [11], where an intrusion is

> any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource.

and the one of [1],

> the potential possibility of a deliberate unauthorized attempt to
> - access information,
> - manipulate information or,
> - render a system unreliable or unusable.

Although some authors use a very detailed list of different intrusions, I will use the simple one given in [15].

- Misuse intrusion

  This kind of intrusion uses the vulnerabilities in the security tools in order to break into the computers. They can be recognized by the fact that the observed events do not follow any logic. But most of the time, it is easier to use the vulnerability signatures (that is, the set of operations one must use to exploit the vulnerability) to spot misuse intrusions.

- Anomaly intrusion

  What happens when the attacker is using someone else's password? Or if he is a regular user who suddenly misbehaves? The problem here is that proper authentications have been given. Yet something wrong is going on. We will see later what can be done about these.

1.5. Security through Obscurity

Many people think secrecy is important to security. This is often referred to as Security through Obscurity, and means that the less we let others know about the security tools, the more secure they are. Therefore the algorithms and the source code are not published. This sounds like a good idea, but it actually is not.

1.5.1. Why it does not work

The main assumption this idea uses is that if the algorithms and the source code are not available, no one will be able to reverse engineer them. It has been proven wrong so many times that it is surprising some people may still believe in it.

Sooner or later (and it tends to be sooner these days), someone will understand the inner working of both the algorithms and the software. What happens then? Is the security mechanism enforced by this software in any way compromised? It depends. If the actual security was the secrecy of the algorithm, then yes, definitely. If it is found that a weak algorithm was used, or that defects were hidden in the code, again, yes. If not, then the software is just as secure as it used to, not less.

But if there is no gain to keep an algorithm secret, why do it? More importantly: if a weak algorithm is used because the author has not the knowledge to implement proper security algorithm, public review will spot it. This in turn will force the author to remove the weak algorithm, otherwise the software will not be used. The same applies to the source code. Peer review can help to spot and fix the defects that could result in vulnerabilities.

It is in fact surprising that so many people trust softwares that have not been reviewed (either by independent specialists under a **N.D.A**, or by every one as in the Open Source system). As one anonymous administrator once said

> Never trust a program you don't have the source code for.

1.5.2. When it could work

But secrecy could work sometimes. As a matter of fact, passwords are kept secret in the head of the users (no other place is appropriate). So where is the border between the useful secrecy and the harmful one?

The passwords' case is quite simple. We saw that secrecy in software was not efficient because of reverse engineering. But there is no way to investigate one user's brain that way (at least, not yet). So hiding a secret where it cannot be investigated is good. A binary only software cannot hide anything.

Another important issue is time. The reverse engineering process can take some time (but that time can be as short as one day). Therefore using such an approach for timely protection can be effective, maybe more than effective than a costly but more robust solution.

Finally, secrecy can be used to confuse attackers, or increase the risk of attacks. For instance, configuring a server so that it uses another port would force any attacker to probe all the ports to find it. With modern scanners this is not hard to do. But monitors could have been put on other ports, and the scanning will be seen. Any good cracker knows it and would not take the risk.

1.6. Softwares' Ecology

Softwares behave like living beings in some respects. When a vulnerability is found in a specific tool, all systems using that tool are at risk. Vulnerabilities are just like predators for a given species. Software's code can also been seen as **DNA**, with families and lineage. Using software coming from a single source can be effective from a management point of view, but is a disaster waiting to happen for security. Recent problems with the Melissa macro virus are a good example.

The best way to prevent such an issue is to mix various softwares and operating systems, so that if one part ever becomes vulnerable, everything else will still work.

Another problem is the inter-relationships between different computers in a system. Once one is compromised, the integrity of other computers are at risk. A vulnerability in just one tool is like a wound that could let to infect the whole organism. Unfortunately, the only way to prevent this is to configure each computer so that it does not trust the others, even when located in the same room. This is not a real solution, because of the hassle it would be for users.

1.7. Planning

The best way to prevent intrusion is to be prepared for it. First, the security officer or the administrator should review every security mechanism and ask himself what would happen if they were compromised. In doing so, he can find that the

configuration could be changed to lower permissions or that other tools could be used to reinforce security, without adding any burden for users.

This operation has to be done on a regular basis, just to make sure that the proper balance between security and users' burden has been achieved.

Then he should ask himself what to do when an intrusion has been done. To spot it is of course mandatory. It would be dramatic if an unauthorized user was allowed to use the system, and changes its behaviour without being noticed. A constant or regular monitoring of the system audit trails as well of other system's variables are just a first step in this direction.

The reaction to such an intrusion must be carefully thought. Restauration of business is part of it and numerous books on general resources management will provide methods to plan it.

But they may leave out some important issues: the local laws may define computer intrusion as a crime, so the victim may want to contact law enforcement agencies to bring the attack to justice. When prosecution is not an option (maybe the intruder was operating from another country where there is no concept of computer crime), the law enforcement agencies are still useful to investigate the case, identify the vulnerabilities used in the attack, and possibly provide some hints about the identity of the intruder. His identity can be used to warn his access provider about his wrong doings and most of the time the provider will take some action of his own. So, in all cases, those agencies should be called, even if it is only for their technical knowledge (law enforcement agencies have or will soon have such a knowledge in most European and American countries).

Another issue is public relation: the public (and shareholders maybe) could have concerns about the ability of the organisation to continue its operations. The public relation department of the organisation is the best suited to prevent those concerns, but will only be effective if it was informed.

Finally, sharing any knowledge gained after this attack (for instance, a new vulnerability) with the Security Community should be considered. If every victim shared this information with everyone, others could prevent an attack and the world would be a safer place. This is what is done in the anti-virus industry and it has proven to be good for both this industry and the users. Again, the law enforcement agencies can provide help for this, because they are likely to know where to put this information.

1.8. Users' Education

Users should be part of the solution, not part of the problem. This is also true with security. Viruses and Trojan horses are too often brought in by unsuspecting users. Teaching them about security to increase their awareness is probably the most important aspect of computer security. They should understand the reasons for authentication and know how to create good passwords, how to identify and

deal with suspect contents and more generally how to reinforce security themselves by monitoring the system during their day-to-day activities.

1.9. Conclusion

One important aspect of computer security to keep in mind is that perfect security is a myth. As Professor Gene Spafford once said:

> " The only system that is truly secure is one that is switched off and unplugged, locked in a titanium safe, buried in a concrete vault on the bottom of the sea and surrounded by very highly paid armed guards. Even then I wouldn't bet on it. "[1]

One could argue that this system is not very useful. It is rather hard to use.

So no matter what we do, the resources are always at risk. This is no different than any other kind of resources. A building could always be destroyed by an earthquake (even if it is unlikely). We can reduce risk, not suppress it. The lower the risk, the higher we pay.

[1] I use to think that a destroyed system was quite secure, until I learned that a hard-disk smashed to pieces, although unusable, could give back huge amounts of information.

chapterSecurity's Tool-box

There is a wide variety of tools a security officer can use to increase the security level of his system. The following classification is an attempt to present these tools in a short and informative way.

1.10. Static Security

Static security includes all tools whose purpose is to block attacks by protecting the security policy. These tools are said to be static because they do not handle intrusions, they just try to prevent them.

1.10.1. Operating System

Any good operating system must provide a basic authentication and security mechanisms. Multi-users operating systems offer some kind of property and access permissions concept. On these systems, one user must authenticate himself before he can use the computer.

The kind of operating system should be chosen according to security requirements. For instance, if a very high level of security is needed, a system that can produce detailed audit trails is better than one that cannot.

It should be noted that most of the time an operating system as shipped by the producer is very insecure. All the default configuration parameters must be reviewed and possible changed. There is a trend towards more sensible defaults, but double-checking is always a good idea[2].

It is important to remember that passwords have a limited impact when remote login is allowed. This is because the default method to send passwords is in plaintext, and that it is very easy to listen to any broadcast media (such as Ethernet) for specific information (such as passwords).

1.10.2. Cryptography

Cryptography has many uses in security. Here is a short list of them:

- one-way encryption[3] of passwords, to prevent their recovery by anyone.
- encryption over insecure channels.
- encryption of data to raise the level of security (so that even if an intruder has access to a file, he still cannot read it).

[2]The only sensible defaults are to close everything and to permit console access to the sysadmin only. It takes longer to configure a system that way, but the result is much better.

[3]One-way encryption is a function that cannot be inversed.

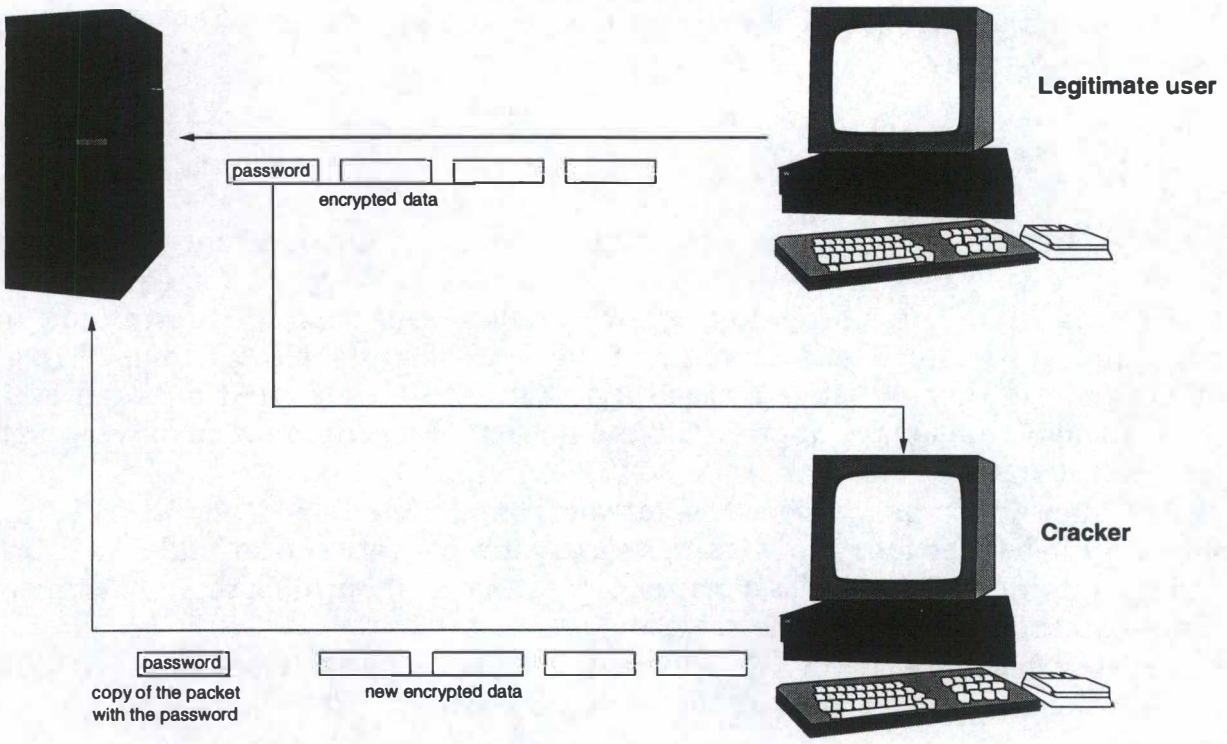

Figure 1.1.: Replay attack scenario

- non-repudiation, with signatures, to make sure any action can be traced back to their author.
- prevention of corruption or modification of data, either by signing the data, or a stamp (produced by a hash function).

Cryptography should be used with care. Its only goal is to make it hard to understand what has been encrypted without the proper key. The following scenario shows a case where cryptography is totally useless (see figure 1.1):

- the communication between the server and the user's computer is encrypted using the server's public key,
- the cracker records the encrypted communication,
- he connects to the server, and when asked for the a password, simply sends the copied packet where this passwords must be.

This is known as the Replay Attack. It can be done because the cracker has not need to decrypt the password. If a new key is used for each new connection, this attack is useless.

Correctly used, cryptography can greatly increase the security a of system. Generally, it should be used for every remote connection.

It is rather unfortunate that there so many political obstacles to the widespread use of cryptography.

1.10.3. Firewall

Firewalls are most often seen as the last line of defense (which they are not) because of their power. They are used to filter network packets based on some properties of these packets. These properties include origin, destination (**IP** address and/or port), protocol, nature, and contents. Some firewalls are able to group **IP** fragments and can then investigate the **TCP** packet. More elaborated ones can even understand a broad range of protocols and make their decision based on very high level consideration.

The filtering can be to reject, log, forward, duplicate or any other action. Thus firewalls can be used for various tasks, such as gateway (between an hidden network and the Internet). So much, as a matter of fact, that a whole book could (and many times had been) devoted to the subject.

Yet that flexibility is also a drawback. It is quite easy to misconfigure a firewall and prevent legitimate traffic, or allow data to leak.

1.10.4. Vulnerabilities Scanner

Vulnerabilities scanners are a very important kind of tools that everyone serious about security should use often. They can detect various problems, either functional defects in software or misconfigurations.

They should be used even for no other reason than because intruders will use them against the system.

Of course, the main problem with them is that they must be updated to be effective. And they only include known vulnerabilities. Even though it may seem obvious, it should be kept in mind that some people may know about a vulnerability for a long time before it becomes public.

1.11. Active Security

The tools presented so far try to prevent intrusions but are rather useless once an intruder has found his way into the system. The following class of tools deals with this.

An intrusion is an hide and seek game. The security officers, helped by the system, try to find the intruder, while the later do his best to leave no trace. In this game, the timing is very important. Being able to detect an intrusion as soon as it started is always better than having to understand later what happened.

1.11.1. Entrapment System

The security officer has an edge over intruders because he knows how the system is configured. If he uses vulnerabilities scanners (and he should), he also knows where the system can be attacked.

Entrapment systems are programs that reproduce a vulnerability, but that are not exploitable. Once installed on the system, they will respond to scanning the same way the vulnerability they simulate. They have the same *signature* as the vulnerability.

If an intruder tries to exploit them, he may think he has found a breach but the programs will just record his activities (and other parameters). It is ideal to catch *script kiddies*[4].

They are different from general Intrusion Detection Systems (see below) because of their relatively limited area of action.

1.11.2. Intrusion Detection System

Finally, what can be done when everything else has failed? The intruder may know the configuration of his target (including the traps), he may have a password, and is about to do some damage.

Intrusion Detection Systems take this possibility into account. They are the last line of defense of a system. I include virus scanners in this category.

The next chapters are entirely dedicated to these tools.

[4]Wanabee crackers who lack the elementary knowledge about security and who use cookbook methods to break in.

2. Intrusion Detection Systems

Static security is like a door. It will only stop the opponent long enough so that the opponent will be spotted. This is what IDS are for.

This chapter uses case studies to hi-light the specific features of each kind of system. The next one introduces a specific architecture.

2.1. Kind of threats

Here we return to the kind of intrusions as defined in [15], and outline the nature of IDS targeted at these intrusions.

2.1.1. Misuse

Misuse is plain violation of security policy. That is, the exploitation of a vulnerability is the system. Of course, vulnerabilities should be fixed as soon as they are found, but sometimes it cannot be done. For this reason, many IDS have been designed to detect such violation.

What kind of information is available? A vulnerability can be exploited by a sequence of actions. Once the vulnerability has been properly studied, this sequence can be known ([15] refers to it as the *signature* of the vulnerability).

Different operating systems generate different amount of audit events for each action they carry on. The range is from nothing to a detailed list of every parameter of the operation. So a vulnerability can be formalized as the list of audit events it generates.

2.1.2. Anomaly

Anomaly intrusion is more vicious. In this case, the intruder has a legitimate password (maybe the intruder is a legitimate user), so he does not have to break the system.

How can we say that something wrong is going on? The only way to detect that kind of intrusion is to detect a variation in the behaviour of the user (from the system's point of view, anyone with a legitimate password is a legitimate user).

Here are the main techniques to detect these anomaly intrusions.

Statistical approach

A profile is defined by a set of measures. Periodically, the system generates a value that measures the abnormality of the profile. This value is a function of the abnormality values of each measures in the profile. For instance a simple function could be a weighted sum of the squares of the individual abnormality values.

Such an approach is interesting because statistics is a well studied area, whose complete tool box can be used here.

But there are some drawbacks:

- the order of events is irrelevant for the profile
- gradual modifications of the profile can change it to allow behaviour that used to be regarded as abnormal
- the threshold above which an anomaly is considered intrusive may be hard to define. If it is too low the number of false positives will rise. If it is too high the number of false negatives will rise.

More information on such systems can be found in [19].

Bayesian Classification

IDS using Bayesian classification are able to determine the most probable numbers of classes and to calculate the probabilistic membership function of each datum (element of data) in the classes (see [3]).

This is a very recent approach and it has not yet been extensively tested.

Neural Networks

In this case, the neural network is trained on a sequence of commands. The input of the net is the current command, and the past w commands. Once the neural network is trained on a set of representative command sequences of a user, it is considered to be the profile of the user. The fraction of incorrectly predicted commands represents the variance of the user behaviour from his profile ([7]).

Neural networks are known to cope well with noisy data. But they can be very slow to converge.

2.2. IDS Architectures

There are severals way to collect the data for an **IDS**. Each offers a balance between scaling, and the precision of the information.

2.2.1. Host based IDS

A host base **IDS** monitors just one computer. This means the events it collects are all generated on that computer.

An instance of that kind of **IDS** is **ASAX**. This system is described in [21] (there are many other documents. Please see the bibliography).

ASAX was a research project of the Computer Department of the Notre-Dame de la Paix University, and Siemens S.A, Namur, Belgium. It is targeted on misuse vulnerabilities and uses a Turing-complete language (RUSSEL) to describe vulnerability signatures. This language can be considered as a query language on the audit trails.

Audit trails are first translated by a **Format Adaptor** into a portable format before they are feeded to the RUSSEL engine. The later activates rules at the initialisation, and these rules can activates other rules on specific events. The set of activated rules represents potential attacks that the engine investigates. The engine has been designed so that the activation of a rule is just an allocation of memory (from the heap) for its parameters.

Static auditing (verification of the configuration) can be done in real time by **ASAX**. It uses a declarative language to mimic the reasoning of an attacker trying to find holes in the configuration. The result is a facts database that can be used to update the rules. The RUSSEL engine triggers the reasoning process when an audit record relative to the modification of a security configuration file is encountered.

Experiments show that it could be used in real time on real life systems with minimal burden.

Disadvantages

- A single host is most of the time nothing but a small part of a larger system. Any intruder will target the system as a whole rather than a host (this is a general drawback of every host-based **IDS**).
- To write attack patterns specifications, one has first to learn RUSSEL, while it may be easier with other tools.
- A monolithic **IDS** is the first target of an intruder.

2.2.2. Multi-hosts based

A network is a bit more than the sum of its hosts. An intrusion can be invisible when seen from any host, but obvious from the network.

But the hosts are still the places where things happen. This is were the audit trails are generated.

Therefore, one must define a way to collect information on hosts and to process it globally.

Distributed ASAX, described in [21] and [22] implements this simply. On each host, the audit processing is used to filter interesting events. An **Audit State Controller** can be used to alter the granularity level used by the auditing mechanism. The **Format Adaptor** translates the audit events for the **Evaluator**.

Filtered events are then sent to a central host where a second level processing can be done and where the **Console** is running. This Console is used to configure and control the whole system. The **Central Evaluator** is in no way different than the other evaluator.

The communication is implemented with **PVM**.

It is obvious that this is a natural extension of **ASAX**, whose independence to the format of audit trails allows such a multi-levels analysis.

2.2.3. Network based

It seems that a multi-hosts based **IDS** can monitor every aspects of a system. That is, as long as the goal of the attacker is to break *into* the system. But **denial of services** do no require such an intrusion.

As a matter of fact, there are several kind of attacks that can only be discovered by monitoring the network. But traditional tools are not able to process such an amount of data. A new kind of **IDS** located between the firewall and the Internet is used for this purpose.

Shadow ([13]) is a **IDS** specifically designed for network analysis. It was able to spot new attack patterns such as coordinated probings and exploits from multiple hosts (possibly from different networks or even countries).

The setup of **Shadow** is simple:

- a **sensor**, located between the firewall and the Internet collects information on packets (by using **tcpdump**).
- inside the firewall, an **analysis station** performs the actual monitoring.
- the only way to access the **sensor** through the firewall is by using a secure shell to copy the data between the **sensor** and the
- the **analysis station** is configured to fetch the data from the **sensor** every hour.

On the **analysis station**, a number of filters is used to find attack patterns in the data. The data is kept for several days, so it is possible to investigate suspicious activities that were not initially seen.

Evaluation

A network-based **IDS** provides valuable information about the attackers. It can detect various probes and forged packets. Moreover, when a computer suddenly crashes, the only way to understand what happened is to used the network data.

But the analysis is very hard. It requires a deep understanding of the protocols. Any mistake in the configuration of the **analysis station** will result in very high number of false negatives or false positives.

Finally, the **sensor**'s location (outside the firewall) allows attacks from the inside.

2.3. Reactive or Proactive IDS

How much reactive or proactive an **IDS** should be? My answer is that an **IDS** should not perform any kind of actions other than raising an alarm. This is based on the following three step reasoning:

1. any **IDS** has false positives.
2. in case of false positive, a legitimate user (and possibly a customer) will be the target of the reacting **IDS**.
3. " the last thing you want is to blow away a legitimate customer. "[25]

A more elaborated answer can be found in [24].

3. Autonomous Agents

It is easy to see a major problem with monolithic intrusion detection systems: once the program stops, there is no monitoring anymore. An intruder knowing that such an **IDS** is in operation on the computer he wants to attack is likely to do his best to fool or crash the **IDS**.

Other kinds of structure should be looked for to solve this problem. looked for. [5] proposes the use of autonomous agents. By agents, we mean

> a group of free-running processes which can act independently of each other and the system. ([5])

The basic architecture should look like in figure 3.1. We see that agents communicate with each other and the triggering of an intrusion alarm is the result of a collaboration between agents, each of them raising the level of a global intrusion risk factor.

No agent is mandatory. If one of them fails, the others are still able to monitor the system. The resulting architecture is much more flexible and foolproof than the monolithic design.

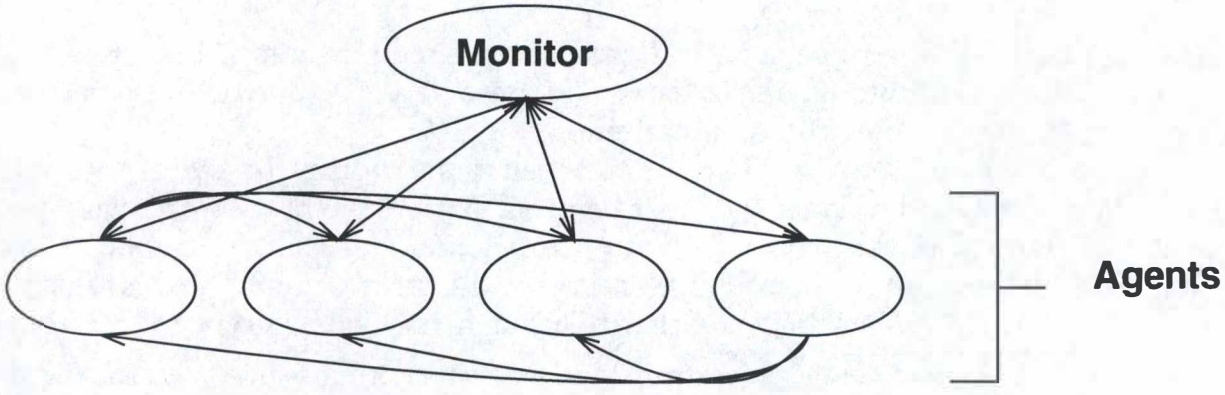

Figure 3.1.: Communication as described in [5]

3.1. Genetic Algorithms

As [5] suggests the use of a genetic algorithm to train the agents, I will first introduce this technique.

The **Genetic Algorithm FAQ** presents it as

> ... a model of machine learning which derives its behavior from a metaphor of some of the mechanisms of evolution in nature.

Its purpose is to find an optimum to some function.

It uses the concept of **population** where **individuals** are described by elementary words (such as chromosomes in real life), the **coding**.

To select promising individuals, one must define a **fitness function** for each possible individual. In fact this is the function we want to optimize.

We start with a randomly filled population, and do the following operations for a given number of **generation**:

evaluation: apply the fitness function to each individual.

reproduction: a fitness-proportionate **reproduction** is then applied. Most promising individuals' coding tends to influence the new individuals more than less promising ones.

mutation: some individuals are stochastically modified. This ensures that no information is lost by always allowing any coding to be inserted.

iteration: the old population is discarded

Although simple, this method can be quite effective. Even though it looks like a random search its use of past solutions to evaluate the present ones allows it to converge quickly.

3.2. Genetic Algorithms in Autonomous Agents

The idea that we could use genetic algorithms to create agents able to recognize attacks patterns without having to hard code these patterns (maybe these patterns are not completely known) is a natural one.

Yet, it may be hard to do. First, we must select the coding. To give agents more flexibility, some kind of language must be used. Each elementary word could be a syntactic element of that language, with place-holders for child elements. Moreover, the syntactic and semantic correctness of the structure must be guaranteed somehow[1]. This can be achieved either by using a very weakly typed language or

[1] It is true that we could use the fitness function to filter out incorrect codings but reproductions and mutations are likely to produce more incorrect codes than correct ones, so the convergence may be too slow.

by using a language syntactically so simple that it is hard to create an incorrect program. The later can be done by using words to represent complex operations. But it means that a lot of effort has to be put in defining these words to make this language as complete as possible with regard to the task[2].

It is clear that such a coding is a tree, not a simple sequence. The reproduction process must be designed according to this structure. And because each node has more importance than the leaves (because the execution is governed by these nodes), a weight-based mixing may be needed.

Finally the agents can only be evaluated when working together. So either we train them to recognize a part of an attack (but then it may be easier to hard code that part) or we train them as a whole (and we have some kind of extended coding composed of each agent's coding). In the later case, what would happen if the genetic algorithm converged so well that the agents stopped working properly if one of them was missing? We would have lost the advantages (at least one of them) of having autonomous agents.

Therefore we see here that even though genetic algorithms sound very interesting to automatically train agents, it is unclear whether this can be achieved.

3.3. $G^A_S SA_T A$: Genetic Algorithms with a twist

Another way to use genetic algorithms is to identify a specific attack. Here, potential attacks are our population and the evolution process is used to select the most likely attack hypothesis. In this case, the attacks patterns are well known, and there is a finite number of them.

This is the basis for $G^A_S SA_T A$ (Genetic Algorithm for Simplified Security Audit Trails Analysis), as described in [20].

3.3.1. Problem

To formalize the problem, we have:

- N_ϵ the number of audit events and N_α the number of potential known attacks,
- AE an $N_\epsilon \times N_\alpha$ attacks-events matrix which gives the set of events generated by each attack.
- R, a N_α-dimensional weight vector, where R_i ($R_i > 0$) is the weight associated with the attack i.
- O, a N_ϵ-dimensional vector where O_i counts the number of events of type i present in the audit trail (O is called " observed audit vector ").

[2] I don't expect such a language to be Turing complete.

- H, a N_α-dimensional hypothesis vector, where $H_i = 1$ if the attack i is present according to the hypothesis, and $H_i = 0$ otherwise.

The problem is thus to find H that maximizes $R \times H$, subject to the constraint $(AE.H)_i \leq O_i$, $(1 \leq i \leq N_\alpha)$. This is a zero-one integer programming problem, thus NP-complete.

3.3.2. Solution

Individuals are hypotheses and the fitness function is

$$F(I_i) = \alpha + \left(\sum_{i=1}^{N_\alpha} R_i.I_i - \beta.T_\epsilon^2 \right)$$

with T the number of audit events for which $(AE.H)_i > O_i$, β the slope of the penalty function, and α a parameter allowing elimination of too unrealistic hypotheses (a negative fitness value is equaled to 0 and the corresponding hypothesis is discarded).

3.3.3. Results

This prototype has been tested on an AIX system, with 24 attacks, 28 audit events and 4 users. The audit trails are filtered in one pass into user-by-user audit vector limited to 30 minutes. The time limit is needed because if the audit trail is too long, the algorithm converges on the N_α-dimensional unit vector.

For this situation, the maximum fitness value converges after about 20 generations and the average fitness value is about 99% of the maximum fitness after 100 generations. So the convergence is very fast.

Of course, the test system is a minimalist one. An experiment on a real system has still to be done.

3.3.4. Disadvantages

While this algorithm is very promising, it has some drawbacks:

- it only works against known attacks
- the absence of event is not taken in account
- multiple realisations of the same attack for a given user are not detected (because of the use of a binary coding)
- multiple attacks can share audit events, in which case the optimum fitness is not reached

- $G^A{}_S SA_T A$ is just an alarm, and the precise attack has then to be located in the audit trails.

4. AAFID: Autonomous Agents For Intrusion Detection

AAFID is an ongoing research project of the **CERIAS** (Center of Education and Research on Information Assurance and Security) of the University of Purdue, West-Lafayette, IN USA. The current version (public-alpha-04) was released September 28th, 1998, but a new version (probably public-alpha-07) should be released in September 1999.

Its purpose is to investigate the strengths and weaknesses of a distributed agents based Intrusion Detection System. It is based on [5], although it does not implement all the ideas developed in this paper (there is no genetic algorithm so far). The current version is documented in [2] (other papers should follow soon and there are many texts included in the package).

It is written in Perl because this language offers a good trade-off between speed and ease of use, and its very dynamic nature (code can be created at run time) makes it ideal for prototypes.

I had the occasion to work on **AAFID** during my internship at Purdue (from September 1998 to January 1999). Here I explain what I did and why.

4.1. Architecture of AAFID

AAFID's architecture is described in [2], and reproduced in short here.

monitors are at the highest level. They provide the user with a graphical interface into the system. A **monitor** is directly called by the user.

transceivers are simply relays between the **monitors** and the **agents** on a given host. Each **transceiver** runs on a single host. The are launched by a **monitor** when the user requests an **agent** to be executed on a host. The **monitor** first checks for a communication channel to a **transceiver** or another **monitor**. If there is one, it is sent a message to load the required entity. Otherwise the **monitor** first call the **Starter** program (using any remote activation system. Currently **AAFID** uses **ssh**.), then set a flag to indicate it is waiting for a connection from that host. The **Starter** instantiates a **transceiver**, contacts the **monitor** (through **TCP**) and redirects its standard input and output to the connection.

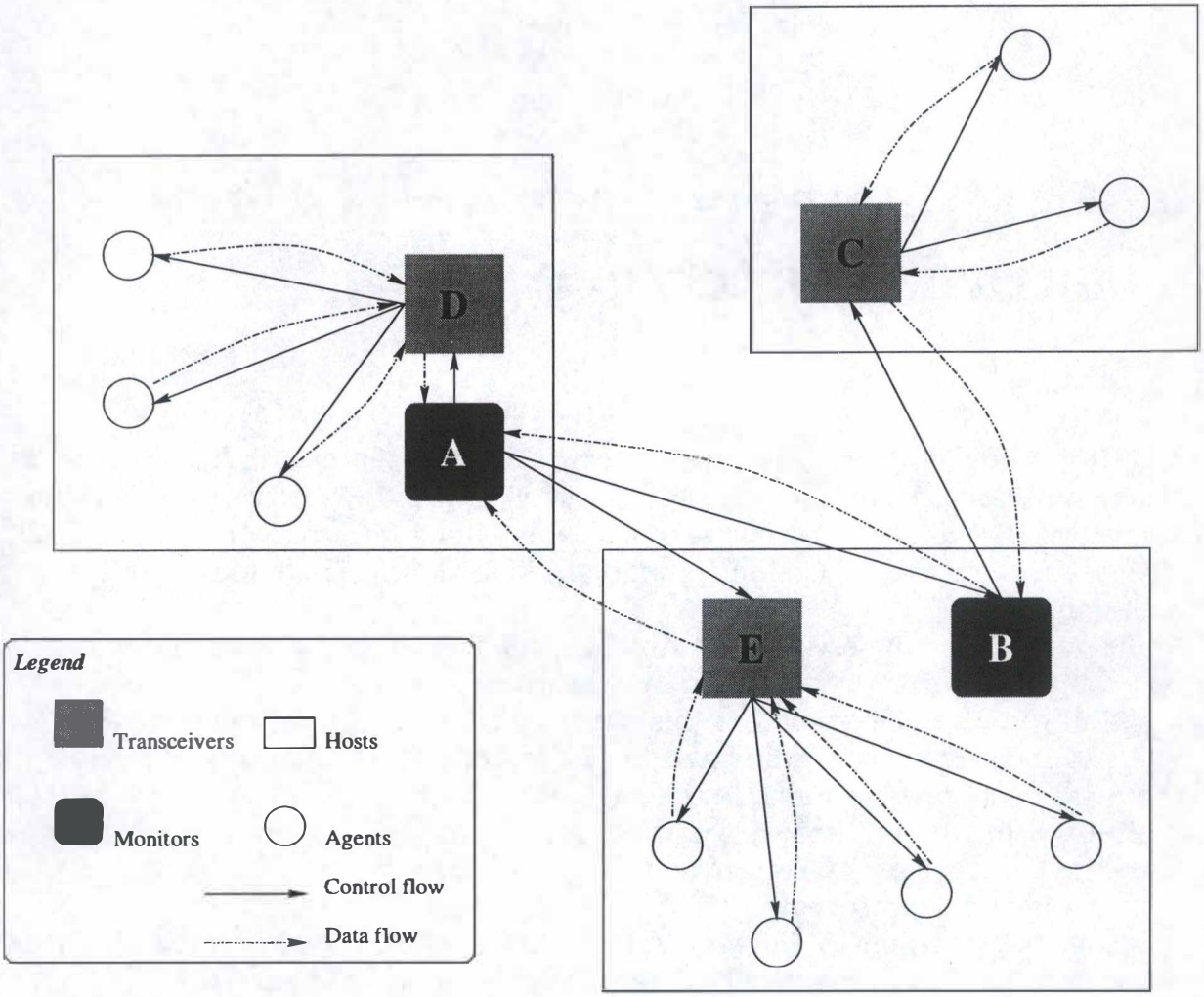

Figure 4.1.: Physical representation of a sample IDS that follows AAFID architecture (as in [2])

Then it runs the **transceiver**, which registers with its **monitor**. The later recognizes the former as the one in which an **agent** has to be started, and sends the appropriate commands to it.

agents are launched by a **transceiver**. The **agent** is first loaded into the **transceiver**, and a new instance is created. Then a process is spawned to run the new instance. The **agents** communicate with their transceiver using Unix **pipes**.

The communication is only done between a parent and its child's (between a given process and the processes its spawned). The figure 4.1 shows the whole structure.

From this figure, it is quite obvious that the communication is vertical and mostly upward and that the hierarchy looks like a pyramid.

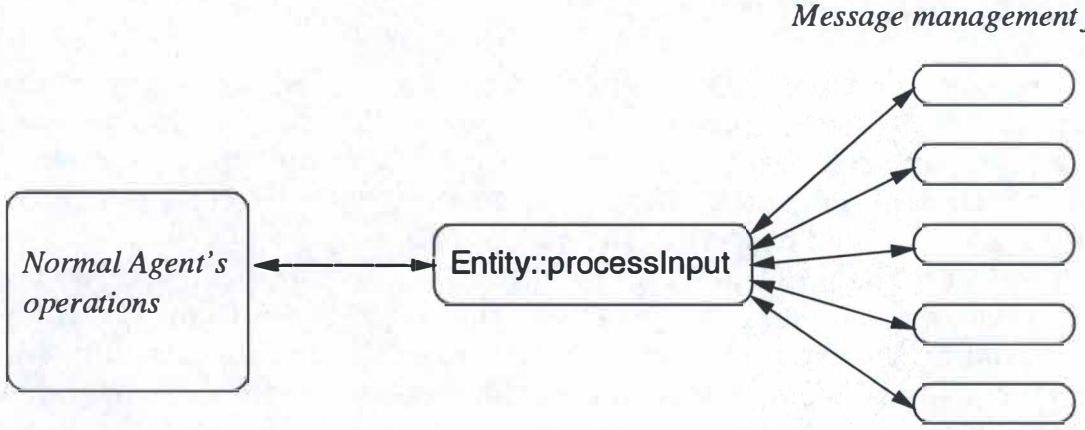

Figure 4.2.: Logical architecture of an AAFID agent

The entities (**agents**, **transceivers** and **monitors**) communicates with each other by sending **messages**. A message is a line of text (a line of text is terminated by a newline character) with a specific format[1].

In its current incarnation, **AAFID** is a framework in which various **IDS** are managed, with a single point of control.

4.2. Architecture of an AAFID agent

An **AAFID agent** is simply a program taking commands from its standard input and sending messages on its standard output[2]. This is useful for debugging purpose. The message management is done in **Entity::processInput**, a huge function handling everything from simple message sending and receiving up to error management, including logging. This means that the semantic (i.e. the format) of messages is defined in this function.

The agent is supposed to call **Entity::processInput** between other operations. These operations perform the actual monitoring. This architecture can be found in figure 4.2.

An **agent** could be written in any language, but the actual execution mechanism is currently tied to Perl.

[1] The format is currently described in an unreleased paper.

[2] When it is created by a **transceiver**, the Unix pipes used for communication are first redirected to the standard input and output.

4.3. Communication is the key

If we compare current's **AAFID** communication (see figure 4.1 on page 26) with what is described in [5] (figure 3.1 on page 19), it is clear that there is no horizontal communications (between agents). In order to provide communication between agents, a complete redesign of the communication mechanism is needed. Actually, this was part of the needed near-term work on **AAFID**.

As we have seen, the problem with the current architecture is the lack of flexibility in the message handler. This function achieve too much to do everything correctly and the management of errors and other unusual conditions is poor at its best. Sometimes the whole hierarchy of agents freezes because one of them does not answer anymore. As [2] puts it:

> If the communication between the entities is somehow disrupted, the system essentially stops working.

My idea was that the use of several layers, with a central module monitoring events and triggering appropriate callback functions on each of them would give better granularity and flexibility.

I designed this module (named the **Reactor**) to be able to respond to time and communication handles' conditions. Afterwards Diego Zamboni added support for file handles, and signals should follow soon.

While designing the module, I kept in mind the saying

> " Mechanisms, no policy. "

My perception is that **AAFID**'s agents can be seen either as small **IDS** on their own, or as members of a group targeted towards a given attack pattern. By mixing these two views, we have an architecture like figure 4.3 on the facing page. In order to attain such a goal, there must be as few restrictions as possible on how to use the communication mechanisms. In particular, the semantic for messages is left to another layer (see below). In short, the **Reactor**'s only purpose is to deliver messages without making any assumption about their format (for instance, the fact that a message is a line of text has been dropped) and to do it with as much reliability and flexibility as possible.

Another module, the **Timer**, provides an interface to Timer objects (see figure 4.4 on the next page) that can manage a tasks list[3]. The tasks can be added with a time (resolution of one second) at which they must be triggered. Removing a task is also supported. The source code can be found at page 41. The version **AAFID** uses includes repeating events, added by Diego Zamboni (not yet release). The actual execution of tasks must be done by the user of a **Timer** object.

[3]When I first wrote the **Timer** module, I called the tasks " events ". This is because I had timeouts in mind. So while I will talk about tasks in this document, the source code still refers to events, although events are only what trigger the tasks.

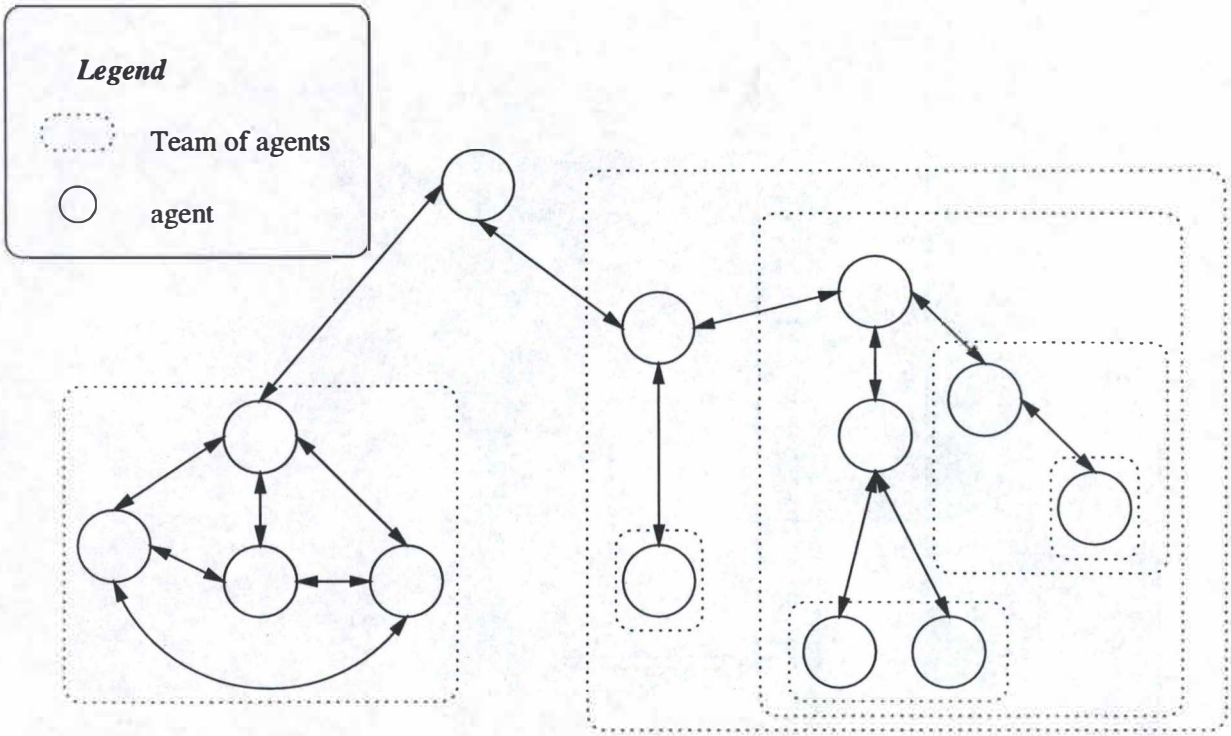

Figure 4.3.: Sample of organisation of agents

Timer
+new(): constructor +add_event(time:integer,func:function reference) +remove_event(time:integer,func:function reference) +get_when(): integer +get_next(): list

Figure 4.4.: Timer Interface

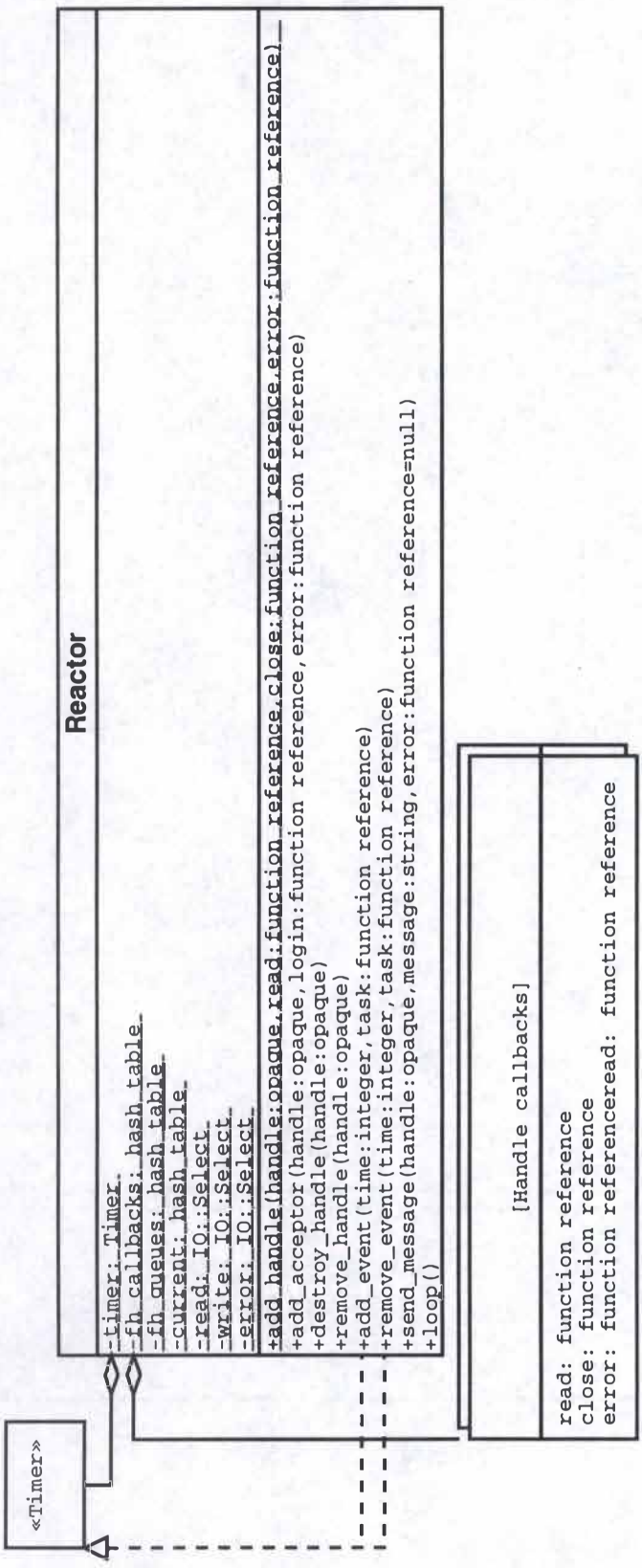

Figure 4.5.: Reactor Interface

The **Reactor** uses the **Timer** and provides an API to insert and remove tasks and communication handles to the system (see figure 4.5 on the facing page). These handles have been chosen because they are an ideal interface to a large number of communication concepts (such as pipes or sockets). The creation of handles is left to the caller of the **Reactor**, therefore any kind of handles can be used, with the exception of files. The management of these (an extension of Diego Zamboni, not yet released) is done by repeating tasks, and other measures are taken on specific conditions (such as when a file is truncated). Acceptor handles (communication handle through which a new connection can be made) are dealt with correctly. The source code can be found on page 44.

4.3.1. Conditions

The **Reactor**'s main work is to check for various conditions (change of state) on the handles, as well as for time events. The work done on each condition is shown below.

Read conditions

On a read condition on one handle, the **Reactor::_read** is called. It is written to be able to manage half-read messages properly. If there is no current message for the given handle, it first blocks to read the 4 bytes size[4] of the new message, then tries to read as much as it can before returning. If the message has not been entirely read, it is first saved in a handle specific variable. If the message is complete, the read callback for this handle is called. If any errors are detected during the operations, the error callback for the handle is called.

Write conditions

Write conditions are only checked on handles for which there is a message waiting to be sent. On a write condition on one handle, the **Reactor::_send** is called. It takes the first entry of the messages queue for the handle, tries to write as much as it can to the handle, then removes the written part out of the message. If the message has been completely sent, it is removed from the messages queue. If there are not any more messages in the messages queue for this handle, the handle is remove from the **writeh** list (this list contains the handle the **Reactor** should check for write condition). If any errors are detected, the error callback for the handle is called.

[4] this is the only place where a blocking read is done. But given the size, it should not be a problem except if less than 4 bytes are sent on purpose.

Error conditions

The error is directly dispatched to the appropriate callback function.

Time events

The function implementing the previous checks for conditions is called to run for at most a given amount of time. This amount is calculated so that the function's timeout occurs when the next set of tasks is to be triggered. When the function returns, the **Reactor** first checks whether any conditions are set on handles then checks if it is time to trigger the tasks.

4.3.2. Sending messages

The message to be sent is first prefixed with its size in network encoding format, then pushed into the messages queue for the given handle and the handle is added to the **writeh** list. The actual sending will be done by **Reactor::_send** on a write condition.

4.3.3. Acceptors

Acceptors are specific handles whose purpose is to receive connection requests.

Acceptors handles are different because they are never written to and they never receive messages. Instead, they can be connected to. From the **Reactor**'s perspective, it looks like a read condition. The **Reactor::_read** function manages this by first checking if the handle is an Acceptor, and if it is, calling the login callback function for this handle. This way, Acceptor handles can be checked with other handles without hassle.

4.3.4. Reactor's loop

The function named **Reactor::loop** is not a real loop. It is rather the place where the **Reactor** waits for anything to happen until the next task is triggered. If there is no task registered, the **Reactor** will simply check for any conditions on each handle in its lists (**readh** for reading, **writeh** for writing, and **errorh** for errors). So if nothing is expected to happen, it exits. The caller of this function should check for termination conditions each time **Reactor::loop** returns.

The architecture of a program using the **Reactor** can be found in figure 4.6 on page 34.

4.3.5. Advantages of the new structure

The **Reactor** does not completely replace the former mechanism, because it lacks the support for relay and logging and procedures activation. But I certainly did not designed it with this features in mind. Its main advantage is to provide easy and flexible support for configurable behaviour for each communication handle. It could be used for various other applications.

Below, I will describe other extensions whose purpose is to provide these features and possibly many more.

4.3.6. Disadvantages

In order to benefit from this structure, the **Reactor::loop** should be used as the application's main loop. This means agents are reactive instead of proactive. The lack of a good multithreads support in Perl makes it hard to mix handles monitoring and **GUI** management, except if the GUI library can manage all the pending events in a single function, returning as soon as no more events are available.

The **Timer** module cannot do hard real time. The tasks will not be triggered sooner than expected, but maybe later (because there is no easy way to suspend the execution at the appropriate time, so if the agent is doing some long processing of inputs, the task can only be activated once the processing is over, when the execution flow is in the **Reactor::loop**. Moreover, the resolution of one second cannot be changed easily.

Finally, moving from a conventional agent to an event-based one is not simple. One has to clearly define the kind of input each handle should manage.

4.3.7. Channels

The **Channel** module's purpose is to provide an easier way to manage handles. By themselves, **Channel** objects are not really interesting. They just give an interface to an handle and use the **Reactor** for most of the work. This interface is described in figure 4.7 on page 35.

The main reason to use them is for the **Services** that can be dynamically added to a **Channel** object to change its behaviour.

A **Channel** can also be an Acceptor. In this case, a login function is supplied to create a new handle for the connection and to register it to the **Reactor**. A function provided by the application is then called on the handle to perform any needed initialisation.

The source code for the **Channel** module can be found page 57.

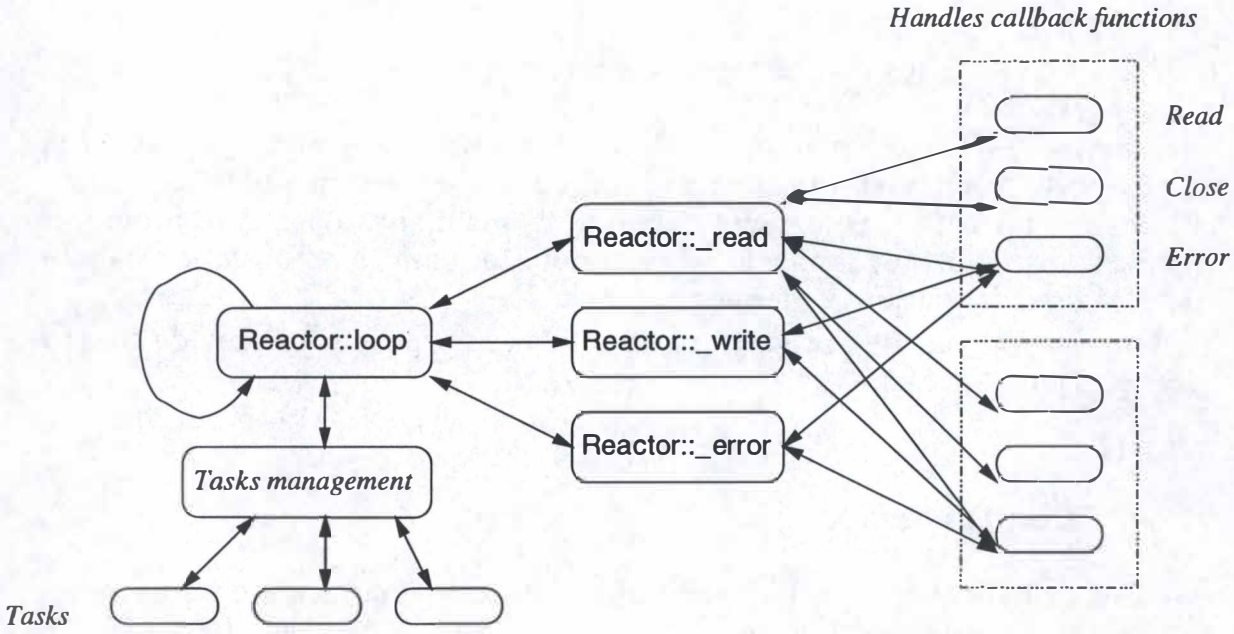

Figure 4.6.: Logical architecture of an AAFID agent with the Reactor

4.3.8. Services

A **Service** is basically an extension for a **Channel** object. It can be called on each sent or received message for any purpose (including a modification). Some **Services** module are included (see figure 4.8 on page 36, and the source code on pages 52, 64, 66 and 69), but these are just samples of what can be done with this mechanism.

Services objects can be used to implement any kind of messages' semantic, on a per-handle basis. Moreover, various semantics can be shared on a given handle. Compared with the **Entity::processInput**'s approach, this is once again more flexible. Since the exploration of the semantics of communication is a medium-term work, I think **Services** can provide a good foundation for it.

Technical description

Services should be able to be added to several **Channels** (the **Services** given as samples are able to do this). This way, a hierarchic configuration (one level for the **Service**, one for each **Channel** and one for each sent or received message) is possible. This configuration (named in the source code **Service options**) is simply a reference to a list of values. It can be given when a message is sent (in this case, it should have as first argument the name of the **Service** it is referring to) or be included in the received message (as a space separated list of values, with the name of the **Service** first, between braces {}, just before the actual message). In both case, the message is an implicit argument.

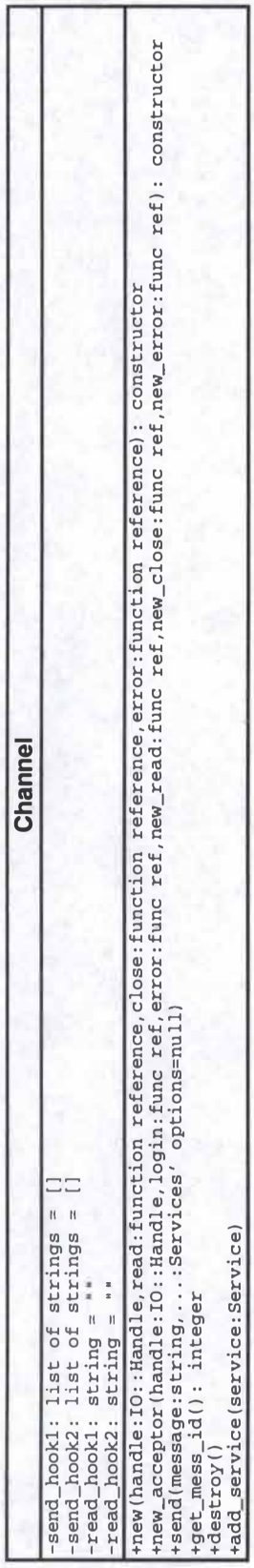

Figure 4.7.: Channel Interface

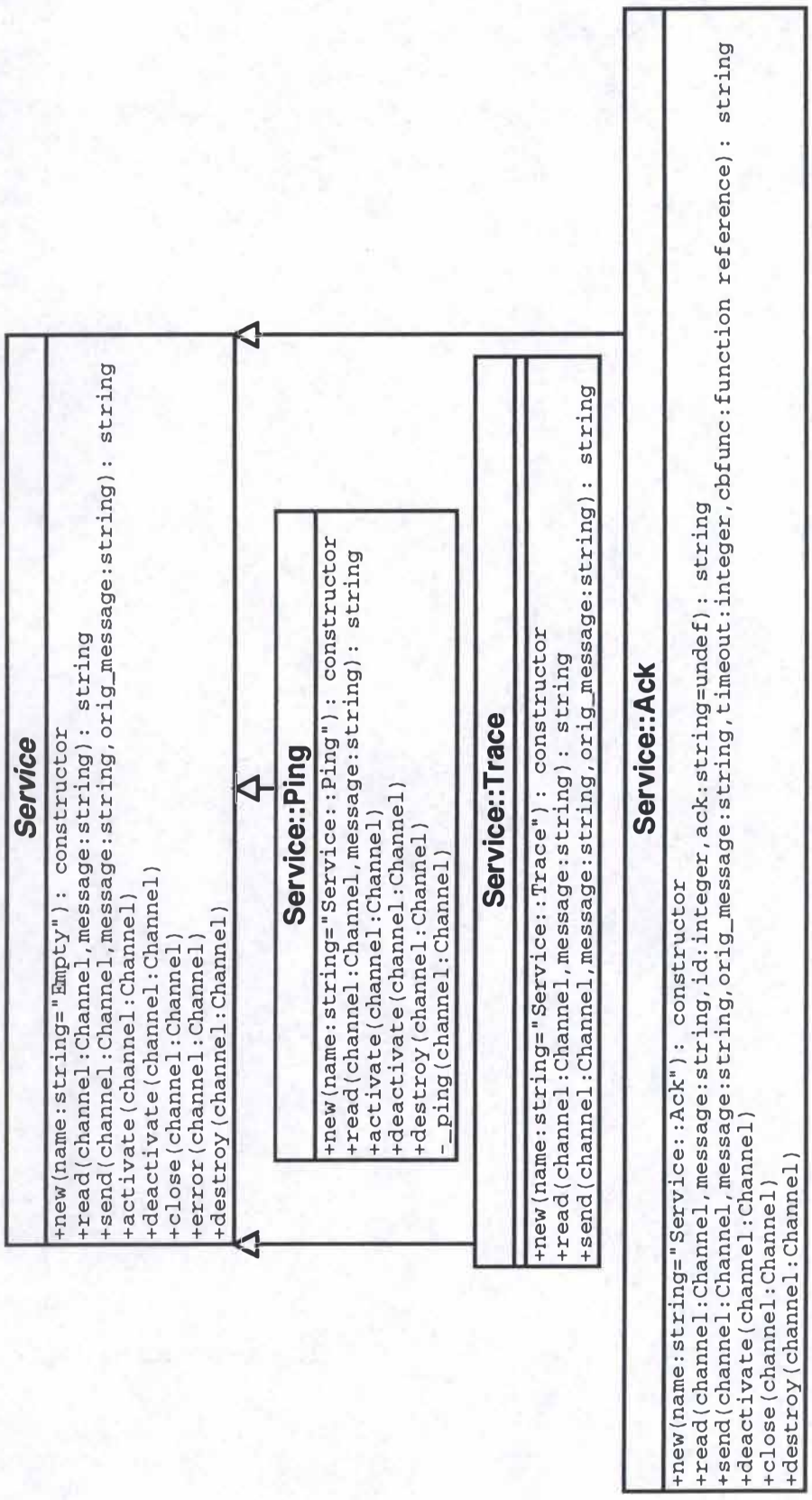

Figure 4.8.: Services Interface

Services can do many things. When they are activated for a given **Channel**, they can request to be called for every message sent through the **Channel** or through the associated handle[5]. They can also be called for a specific message. Likewise, they can be called automatically when a message is received, either for every message coming from the handle, or only for messages that have not been removed by other **Services**. And as we just saw, a specific message can have **Service options** embedded in it.

Sample Services

These samples are just an illustration of the way **Services** work.

Trace simply prints on the standard output the messages sent or received through the **Channel**.

Ping makes sure the peer (the process at the other end of the handle) is still operating by sending **ping** messages to it if nothing is received on the handle for a given period of time. When the peer does not answer, a ping-timeout callback function is called (this function is defined by the **Ping**'s user).

Ack guarantees a message has been properly received by the peer. The only case when a message can be lost is if the peer is suspended or locked in an infinite loop and never returns to the **Reactor::loop**. Of course, **Ping** should warn us if this happens, but it may do it after a very important message (the kind we do not want to loose) has been sent. **Ack** can be used to call a given callback function that takes a message as argument to provide a way to send the message somehow. This is actually a sample of message's semantic.

4.4. What is coming

So far, only the **Reactor** has been included in **AAFID**. According to Diego Zamboni, the other parts look promising but will not make it into the main code for the next public release. Yet I am confident that the whole communication mechanism offers about everything one could want with the proper level of flexibility.

Yet, there are several features that could be added. Several of them (repeating events, support for files) have already been written by Diego Zamboni. One other I think would be worthwhile is the ability to configure the low level nature of a message. Currently it is a string of characters prefixed by its size in network encoding. But other formats can be used by existing **IDS** and this feature will allow them to be integrated in the framework.

[5]There is a difference because **Services** can generate traffic of their own.

Finally, the code does not require any Perl specific features, and could be rewritten in other languages quite easily[6].

[6]As a matter of fact, any typed language would make it easy.

5. Conclusion

Intrusion Detection is without any doubt an essential part of any secure system. To be effective, various approaches must be used to cover all the possibilities.

Practical requirements lead to opposite way. On one hand, small, specific **IDS** are easier to write and to test than one big monolithic application. On the other hand, a single point of control is mandatory.

Therefore, a general framework where **IDS** can cooperate and be managed is ideal. Such a framework should support several kind of **IDS**, each of them with its own communication mechanism and configuration format.

AAFID could be this framework. Even though this was not the original intent, **AAFID** has the potential to integrate different **IDS** into one coherent system. The current work on the flexibility of the communication mechanism is a big step towards this goal. **Services** could be used to transparently translate messages' semantic.

This is a new approach to **IDS**. So far, intrusion detection techniques have been studied, but the cooperation between these techniques is still to be achieved. **ASAX**, with its format independent rules approach, could be the network monitor such a framework is currently missing.

A. Source code

A.1. Timer.pm

The Timer module implements a very simple tasks list and provides a few functions to insert and remove the tasks, and to activate the tasks at the correct time.

```
# Package Timer
# Implements a queue for timed events.
# Frederic Dumont, 1998-1999

=head1 Timer.pm

Timer – Timed event queue

=head1 SYNOPSIS

    use Timer;

    $timer=new Timer;

    $timer->add_event(time()+5,\&some_func);
    $next_events=$time->get_next();
    $next_time=shift @$next_events;
    if ($next_time-time()<=0) {
        foreach (@$next_events) {
            &$_();
        }
    }

=head1 CONSTRUCTOR

=over 4

=item new ( )

The constructor return a timer object.

=back

=head1 METHODS
```

```
=over 4

=item add_event ( TIME, FUNC )

Add the task FUNC with the time TIME (given in seconds since 1/1/1970).

=item remove_event ( TIME, FUNC )

Remove this task that was scheduled at time TIME. Both arguments must
match those given to the add_event method.

=item get_when ( )

Return the time for the next events

=item get_next ( )

Return a list whose first element is the time of the events (that is,
tasks) given as the other elements of the list.

=back

=head1 CALLBACK PROTOTYPE

=over 4

=item task ( )

The task given to the timer cannot take any argument.

=back

=head1 BUGS

I hope not.

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut

package Timer;

use strict;

# Usage:
#     my $timer = new Timer;
sub new {
    my ($pkg)=shift;
```

```
    my ($self)=[];
    return bless $self, $pkg;
}

# Usage:
#    $timer->add_event(time()+$timeout, \&func);
# Rem:
#    use closure if you want to pass arguments to the function
sub add_event {
    my ($self, $timeout, $func) = @_;
    my ($entry)=[$timeout, $func];
    push @$self, $entry;
    @$self = sort { return $$a[0] <=> $$b[0] or $$a[1] <=> $$b[1]; } @$self;
}

# Usage:
#    $timer->remove_event($time, \&func);
# both $time and \&func must match the arguments given at add_event
sub remove_event {
    my ($self, $timeout, $func) = @_;
    @$self=grep { $$_[0] != $timeout or $$_[1] != $func } @$self;
}

# Usage:
#    $next_events=$timer->get_next();
# $next is a reference to a list : ($time, \&func1, \&func2, ...)
# with $time the lowest time entry in the timer, and the corresponding
# function references
sub get_next {
    my ($self) = @_;
    my ($timeout, @entries);
    return [] unless @$self;
    $timeout=$$self[0][0];
    while ($$self[0][0]==$timeout) {
        my $entry = shift @$self;
        push @entries, $$entry[1];
        last unless @$self;
    }
    return [$timeout, @entries];
}

# Usage:
#    $time = $timer->get_when();
# if $time == undef, it means there's no more scheduled events.
# $time is a absolute time that can be compared to time().
sub get_when {
    my ($self) = @_;
    return undef unless @$self;
    return $$self[0][0];
}
```

```
1;
```

A.2. Reactor.pm

The Reactor module is the low-level communication mechanism. It simply holds a list of file handlers and a task list (a Timer object). The main loop polls for events and activate the appropriate callback function for each of them.

```
# Package Reactor
# Poll a set of file handles plus a event queue
# Frederic Dumont, 1998-1999

=head1 Reactor.pm

Reactor - Poll a set of file handles plus a event queue

=head1 SYNOPSIS

=head2 Server code

    use Reactor;
    use IO::Handle;
    use IO::Socket;

    my $fha=new IO::Socket::UNIX(Local=>"toto", Listen=>1);

    sub idle_func {
        print "Idle function\n";
        Reactor::add_event(time()+5,\&idle_func);
    }

    sub login {
        my ($fh) = @_;
        my $nfh=$fh->accept();
        Reactor::add_handle($nfh,\&cb,\&close,\&error);
    }

    sub cb {
        my ($fh, $msg) = @_;
        if($msg eq "quit") {
                print "Received request to quit\n";
                unlink ("toto");
                exit(0);
        }
        print "Msg : $msg\n";
    }

    sub close {
        my $fh=shift @_;
```

```
    print "Connection closed.\n";
    Reactor::destroy_handle($fh);
}

sub error {
    my $fh=shift @_;
    print "Damn! Error on the line.\n";
}

Reactor::add_acceptor($fha,\&login,\&error);
Reactor::add_event(time()+5,\&idle_func);

while(1) {
    Reactor::loop();
}

=head2 Client Code

use Reactor;
use IO::Socket;

$fh=new IO::Socket::UNIX(Peer=>"toto");

while(1) {
    print "Msg? (return for quit)\n";
    $msg=<STDIN>;
    chop $msg;
    last if $msg eq "";
    Reactor::send_message($fh,$msg);
    Reactor::loop();
    last if $msg eq "quit";
}

=head1 Interface

It has been assumed that an application would need only one Reactor.

=cut

package Reactor;

use strict;
use IO::Select;
use vars qw(%acceptor $timer %current %fh_callbacks %fh_queues $readh $writeh $errorh);
use Timer;

# Private variables
my $timer=new Timer;        # the event queue
my %fh_callbacks;           # maps the file handles to the callback functions
my %fh_queues;              # maps the file handles to the send queue
my %current;                # maps the file handles to the current read message
```

```
my $readh=new IO::Select;
my $writeh=new IO::Select;
my $errorh=new IO::Select;

=over 4

=item add_handle ( HANDLE, FUNC, FUNC, FUNC )

Add the given handle to the reactor with the given functions as a callback
(order is 'read', 'close' and 'error')

=cut

sub add_handle {
        my ($fh, $read, $close, $error)= @_;
        $readh->add($fh);
        $errorh->add($fh);
        $fh_callbacks{$fh}->{"read"}=$read;
        $fh_callbacks{$fh}->{"close"}=$close;
        $fh_callbacks{$fh}->{"error"}=$error;
}

=item add_acceptor ( HANDLE, FUNC, FUNC )

Add the given handle as an acceptor to the reactor with the given functions as
callbacks (login and error). An acceptor bypass the message mecanism of the
Reactor and thus has to manage the messages by itself.

=cut

sub add_acceptor {
        my ($fh, $login, $error)=@_;
        $readh->add($fh);
        $errorh->add($fh);
        $acceptor{$fh}->{"login"}=$login;
        $acceptor{$fh}->{"error"}=$error;
}

=item destroy_handle ( HANDLE )

Remove the given handle from the Reactor. This includes the queue of messages
to be sent. That means that once a handle is destroyed, any queued messages
are simply discarded.

=cut

sub destroy_handle {
        my ($fh)=@_;
        $readh->remove($fh);
        $errorh->remove($fh);
        $writeh->remove($fh);
```

```
        delete $fh_callbacks{$fh} if exists $fh_callbacks{$fh};
        delete $fh_queues{$fh} if exists $fh_queues{$fh};
        delete $current{$fh} if exists $current{$fh};
        delete $acceptor{$fh} if exists $acceptor{$fh};
}

=item remove_handle ( HANDLE )

Remove the given handle from the Reactor, but will empty the message queue
first. Once the handle is removed from the Reactor, it can't be listened on.

=cut

sub remove_handle {
        my ($fh)=@_;
        $readh->remove($fh);
        $errorh->remove($fh);
        delete $fh_callbacks{$fh} if exists $fh_callbacks{$fh};
        delete $current{$fh} if exists $current{$fh};
        delete $acceptor{$fh} if exists $acceptor{$fh};
}

=item add_event ( TIME, FUNC )

See add_event in Timer.pm

=cut

sub add_event {
        $timer->add_event(@_);
}

=item remove_event ( TIME, FUNC )

See remove_event in Timer.pm

=cut

sub remove_event {
        $timer->remove_event(@_);
}

=item send_message ( HANDLE, MSG [, FUNC] )

Put the given message (as a string) in the queue for the given handle and
return. The message will be sent whenever it is possible. Blocking is not
supported. The function, if provided, is the callback error function.

=cut

sub send_message {
```

```
        my ($fh, $mesg, $error)=@_;
        $mesg=pack('N',length($mesg)).$mesg;
        push @{$fh_queues{$fh}}, [$mesg, $error];
        $writeh->add($fh);
}

=item loop ( )

The main loop. It has to be called so that I/O and time operations can occur.
If no event is scheduled then the function will only poll the handles and
return if nothing can be done. It should be used in a B<while(1)> loop.

=back

=cut

sub loop {
        my $next_event;
        my $timeout;
        my $fh;
        my $select;
        my ($rset, $wset, $eset);
        while(1) {
                $select=0;
                $next_event=$timer->get_when();
                if(defined $next_event) {
                        $timeout=$next_event-time();
                } else {
                        $timeout=0;
                }
                ($rset,$wset,$eset)=IO::Select::select($readh,
                                                       $writeh,
                                                       $errorh,
                                                       $timeout);
                foreach $fh (@$rset) {
                        _read($fh);
                        $select=1;
                }
                foreach $fh (@$wset) {
                        _write($fh);
                        $select=1;
                }
                foreach $fh (@$eset) {
                        _error($fh);
                        $select=1;
                }
                $next_event=$timer->get_when();
                        # it may have changed
                if ($next_event) {
                        if ($next_event-time()<=0) {
                                my @tmp=@{$timer->get_next()};
```

```
                        shift @tmp;
                        foreach (@tmp) {
                                &$_;
                        }
                }
        }
        return unless $select;
    }
}

=head1 CALLBACK PROTOTYPES

=over 4

=item read ( HANDLE, STRING )

The handle is the one on which the message has been read, and the string is
the message.

=item close ( HANDLE )

The handle is the closed one.

=item error ( HANDLE )

The handle is the one on which an error has been detected.

=back

=cut

# Private methods

# _read is called when a file handle is readable. When the message has been
# read, the callback is called. If an error occurs, it will simply call
# _errors.
# As the reading could be done in several steps, _read stores its state in the
# file handler.
# If anything can be read in a call, but that the connection is closed while
# we're reading, it will be detected at next call.
# The error handling is at best weak. It has almost no recovery of lost or
# partial messages at all.

sub _read {
    my ($fh)=@_;
    if (exists $acceptor{$fh}) {
        &{$acceptor{$fh}->{"login"}}($fh);
        return;
    }
    my $bytes_read;
    my $msg_len;
```

```
        my $mesg;
        my $read_stuff;                 # Did we read anything in this call
        if(exists $current{$fh}) {
                $msg_len=$current{$fh}->{"msg_len"};                    300
                $mesg=$current{$fh}->{"mesg"};
        } else {
                my $buff;
                $bytes_read=sysread($fh, $buff, 4, 0);
                if(defined $bytes_read) {
                        if($bytes_read==0) {
                                &{$fh_callbacks{$fh}->{"close"}}($fh);
                                return;
                        } else {
                                $msg_len=unpack('N', $buff);            310
                                $mesg="";
                                $read_stuff=1;
                        }
                } else {
                        &{$fh_callbacks{$fh}->{"error"}}($fh);
                        return;
                }
        }
        while($bytes_read=sysread($fh, $mesg, $msg_len,
                                length($mesg))) {                       320
                $read_stuff=1;
                $msg_len-=$bytes_read;
                last if $msg_len==0;
        }
        if(!defined $bytes_read) {
                $current{$fh}->{"mesg"}=$mesg;
                $current{$fh}->{"msg_len"}=$msg_len;
                &{$fh_callbacks{$fh}->{"error"}}($fh);
                return;
        } elsif($msg_len==0) {                                          330
                delete $current{$fh};
                &{$fh_callbacks{$fh}->{"read"}}($fh,$mesg);
                return;
        } elsif(!defined $read_stuff) {
                # We did not read anything on this call, so the connection is
                # closed.
                &{$fh_callbacks{$fh}->{"close"}}($fh);
                return;
        } else {
                $current{$fh}->{"mesg"}=$mesg;                          340
                $current{$fh}->{"msg_len"}=$msg_len;
        }
}

# _write is called when a file handle is writable. It will push as much as it
# can in the file handle and update the message queue. If an error occurs, it
# will call the given error callback if it is defined.
```

```
sub _write {
        my ($fh)=@_;
        my $bytes_written;
        my $mesg=$fh_queues{$fh}->[0]->[0];
        my $error=$fh_queues{$fh}->[0]->[1];
        while($bytes_written=syswrite($fh, $mesg, length($mesg))) {
                $mesg=substr($mesg,$bytes_written,-1);
                if($mesg eq "") {
                        shift @{$fh_queues{$fh}};
                        unless (@{$fh_queues{$fh}}) {
                                $writeh->remove($fh);
                        }
                        return;
                }
        }
        $fh_queues{$fh}->[0]->[0]=$mesg;
        if(!defined $bytes_written) {
                &error($fh) if $error;
                shift @{$fh_queues{$fh}}; # we clean the message queue.
                $writeh->remove($fh);
        }
}

# _error just dispatch to the given callback

sub _error {
        my ($fh)=@_;
        if (exists $acceptor{$fh}) {
                &{$acceptor{$fh}->{"error"}}($fh);
        } else {
                &{$fh_callbacks{$fh}->{"error"}}($fh);
        }
}

1;

=head1 BUGS

The module will block to read the size of a message (a 4 bytes long integer).
If no more than 3 bytes are sent, the module freezes. This is not fun to fix.

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut
```

A.3. Service.pm

The Service Module provides the base class for all Services. They are extensions for the Channel objects. They can be activated (called on each message, sent or received). Services can also generate their own messages. The application should not see those.

```
# Package Service
#
# Frederic Dumont, 1998-1999

=head1 Service.pm

Service – provides the base class for extensions to the Channel objects

=head1 SYNOPSIS

Service is supposed to be the base class for all other Services.

=cut

package Service;

use strict;

sub _insert_read_hook;
sub _insert_send_hook;
sub _remove_read_hook;
sub _remove_send_hook;

=head1 CONSTRUCTOR

=over 4

=item new ( [ STRING ] )

Returns a new object. If a name is given, it will be used as the name of the
Service.

=back

=cut

sub new {
        my $pkg = shift @_;
        my $name;
        if (@_) {
                $name=shift @_;
        } else {
                $name=$pkg;
```

```
            use strict;
        }
        my $self={"name"=>$name};
        return bless $self, $pkg;
}

=head1 METHODS

=over 4

=item read ( CHANNEL, STRING, ... )

read takes the channel and the read message to process, and any number of other
arguments of any kind (these are the options). It returns the processed
message, or undef if the message has completely been handled.

=cut

sub read {
        my ($self,$chan,$mess,@options)=@_;
        return $mess;           # The default behaviour is to do nothing
}

=item send ( CHANNEL, STRING, STRING, ... )

send takes the channel sending the message and the message to process, the
original message (the one given to the Channel->send function) and any number
of other arguments of any kind.  It returns the processed message, or undef if
the message has directly been sent.

=cut

sub send {
        my ($self, $chan, $mess, $orig_mess, @options)=@_;
        return $mess;
}

=item activate ( CHANNEL, ... )

activate allows the given Channel to use the Service implicitely. Any number of
other arguments of any kind can be added (these are the default options).

=cut

sub activate {
        my ($self, $chan, @options)=@_;
        if ($self->_insert_send_hook($chan->{"send_hook1"})) {
                $chan->{"read_hook1"}=
                        $self->_insert_read_hook($chan->{"read_hook1"});
        }
}
```

```
=item deactivate ( )

deactivate removes the Service from the list of implicitely called Services of
the given Channel. It always succeeds.

=cut

sub deactivate {
        my ($self, $chan)=@_;
        $self->_remove_send_hook($chan->{"send_hook1"});
        $chan->{"read_hook1"}=$self->_remove_read_hook($chan->{"read_hook1"});
}

=item close ( CHANNEL )

close may be called by the Channel when the handle has been closed. The default
action is to deactivate the Service.

=cut

sub close {
        my ($self, $chan)=@_;
        $self->deactivate($chan);
}

=item error ( CHANNEL )

error may be called by the Channel when there is an error on the handle. The
default action is to deactivate the service.

=cut

sub error {
        my ($self, $chan)=@_;
        $self->deactivate($chan);
}

=item destroy ( CHANNEL )

destroy will be called when the Channel is destroyed.

=back

=cut

sub destroy {
}

1;
```

```
=head1 PRIVATE FUNCTIONS

=over 4

=item _insert_send_hook ( LISTREF )                                    150

_insert_send_hook takes the correct send_hook (1 or 2), add the Service
to it and return 1. If the service is already in it, it returns 0.

=cut

sub _insert_send_hook {
        my ($self, $send_hook)=@_;
        foreach (@$send_hook) {
                return 0 if ($_ == $self->{"name"});                   160
        }
        push @$send_hook, $self->{"name"};
        return 1;
}

=item _insert_read_hook ( STRING [, STRING ] )

_insert_read_hook will take the correct read_hook (1 or 2) and a string to
be added to the read_hook (if null, the name of the Service will be used).
It add the Service to it, and return the new read_hook.                170
The braces {} are added here.

=cut

sub _insert_read_hook {
        my ($self,$read_hook,$string) = @_;
        $string=$self->{"name"} unless $string;
        return $read_hook."{".$string."}";
}
                                                                       180
=item _remove_send_hook ( LISTREF )

_remove_send_hook will remove the Service name from the list reference.

=cut

sub _remove_send_hook {
        my ($self, $send_hook) = @_;
        @$send_hook = grep { $_ ne $self->{"name"} } @$send_hook;
}                                                                      190

=item _remove_read_hook ( STRING )

_remove_read_hook will remove the Service options (the first one if there are
several Service options for the same Service) from the given string and
return the result.
```

```
=back

=cut                                                                    200

sub _remove_read_hook {
        my ($self, $read_hook) = @_;
        if ($read_hook=~/{$self->{"name"}([^{]*)}/) {
                return $'.$';
        }
        return $read_hook;
}

=head1 CLASS FUNCTIONS                                                  210

=over 4

=item extract_service ( STRING )

extract_service first checks for the presence of a Service related data at
the start of the message, and if there is one, removes it and splits it to
an array. It returns an array with the message, then the Service data array
(may be empty).
                                                                        220
=cut

sub extract_service {
        my $mess = shift @_;
        my @service;
        if ($mess=~/^{([^{]*)}(.*)$/) {
                $mess=$2;
                @service=split /\s+/, $1;
        }
        return ($mess, @service);                                       230
}

1;

=back

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>
                                                                        240
=cut
```

A.4. Channel.pm

The application deals mainly with Channel objects. Messages are sent or received through Channels. Any number of Services can be added and activated.

```
# Package Channel
#
# Frederic Dumont, 1998-1999

=head1 Channel.pm

Channel - each Channel object manages a communication channel (hence the name)

=head1 SYNOPSIS

The purpose of a Channel object is to manage a communication channel with
another Channel object through sockets, and to provide several Services (see
below) on this channel.

Services are extensions to the basic send-receive mechanism. Basic Services
include Ping and Ack. Others (such as encryption, configuration) can be added
easily.

=cut

package Channel;

use strict;
use Reactor;
use Service;
use vars qw(%fhmap);

# Private variable

my %fhmap;          # map file handle to the corresponding Channel object
sub 'login;         # login function for acceptors
sub 'read;          # main read function
sub 'close;         # main close function
sub 'error;         # main error function

=head1 CONSTRUCTOR

=over 4

=item new ( HANDLE, FUNC, FUNC, FUNC )

Creates a new Channel object with the given functions as read, close and error
callbacks. The handle must be a proper handle object (see IO::Handle).

=cut
```

```
sub new {
        my ($pkg, $fh, $read, $close, $error)=@_;
        Reactor::add_handle($fh,\&read,\&close,\&error);
        my $self={"handle"=>$fh,  # file handle                              50
                  "read"=>$read,    # read callback
                  "close"=>$close,  # close callback
                  "error"=>$error,  # error callback
                  "services"=>{},   # services dictionary
                  "mess_id"=>1,     # next message id;
                  "send_hook1"=>[], # high level filters (only for msg)
                  "send_hook2"=>[], # low level filters (mandatory on this
                                    # Channel)
                  "read_hook2"=>"", # This string will be added in front
                                    # of each received message.              60
                  "read_hook1"=>"", # This string will be added in front
                                    # of each received message unless
                                    # they are processed by a Service.
                                    # Activated Services can use it to
                                    # called when a message for the
                                    # application is received.
                  };
        $fhmap{$fh}=$self;
        return bless $self, $pkg;
}                                                                           70

=item new_acceptor ( HANDLE, FUNC, FUNC, FUNC, FUNC, FUNC )

Creates a new acceptor Channel with the given function as error callback. The
next three functions are the read, close and error callbacks of new Channels
created by the acceptor. The last one is the init function that will be called
on each new Channel.

=cut
                                                                            80
sub new_acceptor {
        my ($pkg, $fh, $error, $new_read, $new_close, $new_error, $init)=@_;
        Reactor::add_acceptor($fh,\&_login,\&_error);
        my $self={"handle"=>$fh,
                  "login"=>\&_login,
                  "error"=>\&_error,
                  "new_read"=>$new_read,
                  "new_close"=>$new_close,
                  "new_error"=>$new_error,
                  "init"=>$init,                                            90
                 };
        $fhmap{$fh}=$self;
        return bless $self, $pkg;
}

=back
```

```
=head1 METHODS

=over 4                                                              100

=item send ( STRING, .... )

send takes the message to be sent, and a list of Services options. Each Service
option is a (reference to a) list, composed of the Service's name and the
options. A Service can take the message out of the Channel by returning undef
to its send call.

=cut
                                                                     110
sub send {
      my ($self, $mess, @service) = @_;
      my %local_serv;
      my $orig_mess=$mess;
      foreach (@service) {
            $local_serv{shift @$_}=$_
      }
      my %first_serv=$self->_filter_serv(%local_serv);
      # We first select the services that are not in send_hook1 or send_hook2.
      foreach (keys %first_serv) {                                   120
            $mess=$self->{"services"}->{$_}->send($self,
                                        $mess,
                                        $orig_mess,
                                        @{$first_serv{$_}});
            return unless $mess;
      }
      my @local_stack=@{$self->{"send_hook1"}};
      foreach (@local_stack) {
            if (exists $local_serv{$_}) {
                  $mess=$self->{"services"}->                        130
                        {$_}->send($self,
                                   $mess,
                                   $orig_mess,
                                   @{$local_serv{$_}});
                  delete $local_serv{$_};
            } else {
                  $mess=$self->{"services"}->
                        {$_}->send($self,
                                   $mess,
                                   $orig_mess);                      140
            }
            return unless $mess;
      }
      $self->_low_send($mess,$orig_mess,%local_serv);
}

=item get_mess_id ( )
```

```
get_mess_id returns a different integer each times it is called. It may be
used by Services for tagging some messages.

=cut

sub get_mess_id {
        my $self=shift @_;
        return $self->{"mess_id"}++;
}

=item destroy ( )

destroy will remove the handle from the Reactor and clean everything else.

=cut

sub destroy {
        my $self = shift @_;
        delete $fhmap{$self->{"handle"}};
        Reactor::destroy_handle($self->{"handle"});
        foreach (keys %{$self->{"services"}}) {
                $self->{"services"}->{$_}->destroy($self);
        }
}

=item add_service( SERVICE )

Add the given Service.

=back

=cut

sub add_service {
        my ($self, $serv)=@_;
        my $name=$serv->{"name"};
        $self->{"services"}->{$name}=$serv;
}

=head1 ATTRIBUTES

The attributes are not meant to be used by the application. They are normally
reserved for the Services.

=over 4

=item send_hook1

This list is for activated Services that should be called on each message sent
by the application. Each Service can provide a way to deal with default
```

```
options.

=item send_hook2

This list is for activated Services that should be called for each messages on
this Channel. Keep in mind that a Service can generate messages too.

=item read_hook1

This is a string where activated Services can put a Service option (a string
that will be recognized by Services). This string will be added in front of
each message received for the application (but not for the messages already
handled by a Service).

=item read_hook2

This is a string where activated Services can put a Service option. This string
will be added in front of each received message.

=cut

# _login creates a new file handle with accept() and a new Channel with this
# handle

sub _login {
        my $fh=shift @_;
        my $self=$fhmap{$fh};
        my $new_fh=$fh->accept();
        my $new_fh=new Channel($new_fh,
                               $self->{"new_read"},
                               $self->{"new_close"},
                               $self->{"new_error"});
        &{$self->{"init"}}($new_fh);
}

# _filter_serv takes a hash of Services options, and removes those whose name
# is in send_hook1 or send_hook2

sub _filter_serv {
        my ($self,%serv)=@_;
        my %result;
        my $key;
        foreach $key (keys %serv) {
                unless ( (grep { $key eq $_ } @{$self->{"send_hook1"}})
                     or (grep { $key eq $_ } @{$self->{"send_hook2"}} )) {
                  $result{$key}=$serv{$key};
                }
        }
        return %result;
}
```

```
# _service parse the message, extract the first Service data and gives it to
# the proper function.

sub _service {
        my ($self,$mess)=@_;
        my @service;
        ($mess,@service)=Service::extract_service($mess);
        return $mess unless @service;
        $mess=$self->{"services"}->{shift @service}->read($self,
                                                $mess,
                                                @service);
        return $mess;
}

# _read is called by the Reactor. It checks for services calls, and if the
# message has not been handled by a Service, it calls the read callback.

sub _read {
        my ($fh,$mess)=@_;
        my $self=$fhmap{$fh};
        $mess=$self->{"read_hook2"}.$mess;
        my $new_mess=$mess;
        do {
                $mess=$new_mess;
                $new_mess=_service($self,$mess);
        } until ($new_mess eq $mess);
        if ($mess) {        # if the message has not yet been handled
                            # by a Service
                $mess=$self->{"read_hook1"}.$mess;
                $new_mess=$mess;
                do {
                        $mess=$new_mess;
                        $new_mess=_service($self,$mess);
                } until ($new_mess eq $mess);
                &{$self->{"read"}}($mess);
        }
}

# _close and _error are called by the Reactor. They find the appropriate
# channel, and call the appropriate callback

sub _close {
        my ($fh)= shift @_;
        my $chan=$fhmap{$fh};
        &{$chan->{"close"}}($chan);
}

sub _error {
        my ($fh)= shift @_;
        my $chan=$fhmap{$fh};
        &{$chan->{"error"}}($chan);
```

```
}

# _low_send send the message through the low level (and mandatory) filters
# and finally to the Reactor. A Service can remove the message from the
# Channel by returning undef from its send call.

sub _low_send {
        my ($self, $mess, $orig_mess, %local_serv) = @_;
        my @local_stack=@{$self->{"send_hook2"}};
        foreach (@local_stack) {                                        310
                if (exists $local_serv{$_}) {
                        $mess=$self->{"services"}->
                                {$_}->send($self,
                                        $mess,
                                        $orig_mess,
                                        @{$local_serv{$_}});
                        delete $local_serv{$_};
                } else {
                        $mess=$self->{"services"}->
                                {$_}->send($self,                       320
                                        $mess,
                                        $orig_mess);
                }
                return unless $mess;
        }
        Reactor::send_message($self->{"handle"},
                        $mess,
                        $orig_mess);
                                # no error callback here. We'll deal
                                # with this through the normal error    330
                                # callback.
}

1;

=back

=head1 CALLBACK PROTOTYPES

=over 4                                                                 340

=item read ( STRING )

read does not take an Channel argument because it is the boundary between the
Channel and the application. If a message must be handled internally by the
Channel, it is best left to a Service (read callbacks for Services have a
Channel argument).

=item close ( CHANNEL )
                                                                        350
It should either call the close function of each Services with
```

```
foreach (@{$self->{"services"}}) { $_->close($self); }

or destroy itself (that will automatically call destroy on the Services).

=item error ( CHANNEL )

If the error cannot be fixed, the same operations as for close should be done.

=item init ( CHANNEL )

This is where the new Channel can be initialized, and Services added and
activated.

=back

=head1 BUGS

There is no intern configuration mechanism for the Services, which means that a
Channel could receive an unmanageable message (that is, it has not the correct
Service to handle it, making it crash). An external Configuration Service can
fixe this.

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut
```

A.5. Service::Trace.pm

The Service::Trace module is an example of a somewhat useful Service. Once activated, it will print on the standard output each message received for the application.

```
# Package Service::Trace
#
# Frederic Dumont, 1998-1999

=head1 Trace.pm

Trace - provides an easy way to print each messages for or from the
application.

=head1 SYNOPSIS

Trace will print on the standard output each message for or from the
application.
```

```
=cut

package Service::Trace;
use Service;
@ISA=("Service");
use strict;

=head1 METHODS

=over 4

=item read ( CHANNEL, STRING )

read takes the channel and the read message to process. It prints the message
on the standard output then return the message.

=cut

sub read {
        my ($self,$chan,$mess)=@_;
        print "Trace:",$mess,"\n";
        return $mess;
}

=item send ( CHANNEL, STRING, STRING )

send does the same thing as read, but for outbound messages.

=cut

sub send {
        my ($self,$chan,$mess, $orig_mess)=@_;
        print "Trace:",$orig_mess,"\n";
        return $mess;
}

=item activate ( CHANNEL )

Each message will be printed on the standard output. It uses the
Channel->read_hook1 to be called on each read message, and the
Channel->send_hook1 to be called on each sent message.

=cut

=item deactivate ( )

deactivate removes the Service from the list of implicitely called Services of
the given Channel. It always succeeds.

=cut
```

```
1;

=back

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut
```

A.6. Service::Ping.pm

The Service::Ping module shows how to use the _low_level function of the Channel objects. It can also be used to verify that the peer is still alive (that is, not frozen).

The implementation is somewhat complex due to the fact that this object takes any kind of message received from the peer as a proof that it is still alive. So it uses Channel-¿read_hook2 to be called each time a message arrives.

```
# Package Service::Trace
#
# Frederic Dumont, 1998-1999

=head1 Ping.pm

Ping - ping the peer to detect bad behaviour

=head1 SYNOPSIS

The Ping Service can be used to detect frozen peers. Once activated, it pings
the peer, and wait for an answer. If none are given, it triggers the timeout
callback. If an answer is received, it pings again.

=cut

package Service::Ping;
use Service;
@ISA=("Service");

use strict;

=head1 METHODS

=item activate ( CHANNEL, INT, INT, FUNC )

The Ping object will start to send ping messages to the peer, wait for an
answer, and ping again. If no answer comes back before Ping->timeout seconds,
the timeout function is called. If anything comes down the Channel, the Ping
```

service will takes this for an evidence that peer is alive, and will be put on hold for an idle number of seconds before sending the next ping.

The arguments are the Channel on which it is activated, the idle time (the time between a ping reply and the next ping), the timeout time (the time to wait before triggering the timeout callback function) and the timeout callback function.

=cut

```
sub activate {
        my ($self, $chan, $idle, $timeout, $cbfunc)=@_;
        $self->{$chan}->{"idle"}=$idle;
        $self->{$chan}->{"timeout"}=$timeout;
        $self->{$chan}->{"cbfunc"}=$cbfunc;
        $self->{$chan}->{"ping_func"} = sub { _ping($self,$chan); };
        $self->{$chan}->{"next_ping"}=time()+$idle;
        $chan->{"read_hook2"}=
                $self->_insert_read_hook($chan->{"read_hook2"},
                                         $self->{"name"}." REPL");
        Reactor::add_event($self->{$chan}->{"next_ping"},
                           $self->{$chan}->{"ping_func"});
}
```

=item deactivate ( )

The Ping object will not ping anymore.

=cut

```
sub deactivate {
        my ($self, $chan) = @_;
        if (exists $self->{$chan}->{"next_timeout"}) {
                Reactor::remove_event($self->{$chan}->{"next_timeout"},
                                      $self->{$chan}->{"cbfunc"});
        }
        if (exists $self->{$chan}->{"next_ping"}) {
                Reactor::remove_event($self->{$chan}->{"next_ping"},
                                      $self->{$chan}->{"ping_func"});
        }
        delete $self->{$chan};
        $chan->{"read_hook2"}=
                $self->_remove_read_hook($chan->{"read_hook2"});
}
```

=item read (CHANNEL, STRING [, STRING])

The first string (the message) is returned, and the second string determines whether we should send a ping-reply or not.

=cut

```
sub read {
        my ($self, $chan, $mess, $option) = @_;
        if (!defined $option) {
                my $mess="{".$self->{"name"}." REPL}";
                $chan->_low_send($mess,$mess);
        }
        if (exists $self->{$chan}->{"next_timeout"}) {
                Reactor::remove_event($self->{$chan}->{"next_timeout"},
                                $self->{$chan}->{"cbfunc"});                    90
                delete $self->{$chan}->{"next_timeout"};
        }
        if (exists $self->{$chan}->{"next_ping"}) {
                Reactor::remove_event($self->{$chan}->{"next_ping"},
                                $self->{$chan}->{"ping_func"});
        }
        $self->{$chan}->{"next_ping"}=time()+$self->{$chan}->{"idle"};
        Reactor::add_event($self->{$chan}->{"next_ping"},
                        $self->{$chan}->{"ping_func"});
        return $mess;                                                           100
}

sub destroy {
        my ($self, $chan) = @_;
        $self->deactivate($chan);
        delete $self->{$chan};
}

# _ping: the low level function used in closure to send pings.
                                                                                110
sub _ping {
        my ($self, $chan)=@_;
        my $mess="{".$self->{"name"}."}";
        my $time=time()+$self->{$chan}->{"timeout"};
        $self->{$chan}->{"next_timeout"}=$time;
        Reactor::add_event($time,$self->{$chan}->{"cbfunc"});
        $chan->_low_send($mess,$mess);
}

1;                                                                              120

=head1 CALLBACK PROTOTYPES

=item timeout ( )

The timeout callback function takes no arguments.

=back

=head1 AUTHOR                                                                   130
```

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut

A.7. Service::Ack.pm

The Service::Ack can be called on sent messages to request that a confirmation of reception by peer be sent. It is useful because some messages are too important to be lost if the crash of the peer is discovered after the messages have been sent. With Service::Ack, those messages can be sent to an alternative location.

```
# Package Service::Ack
#
# Frederic Dumont, 1998-1999

=head1 Ack.pm

Ack - send a message with a demand for confirmation

=head1 SYNOPSIS

                                                                    10
Ack is invoked in Channel->send to request that a confirmation be sent by the
peer. If that confirmation is not received within a given amount of time,
a timeout function is called.

=cut

package Service::Ack;
use Service;
@ISA=("Service");
                                                                    20
use strict;

=head1 METHODS

=over 4

=item read ( CHANNEL, STRING, INT [, STRING ] )

read takes the channel, the read message to process, and the message id. The
optional string indicates this is a answer. If not, it replies with this    30
message id.

=cut

sub read {
        my ($self,$chan,$mess, $id, $ack)=@_;
        if ($ack) {
```

```
            if (exists $self->{$chan}->{$id}) {
                Reactor::remove_event($self->{$chan}->
                                        {$id}->{"timeout"},
                                      $self->{$chan}->
                                        {$id}->{"timeout_func"});
                delete $self->{$chan}->{$id};
            }
        } else {
            my $new_mess="{".$self->{"name"}." ".$id." ACK}";
            $chan->_low_send($new_mess,$new_mess);
        }
        return $mess;
}

=item send ( CHANNEL, STRING, STRING, INT, FUNC )

send uses the provide function as timeout callback, and the integer as the
timeout period.

=cut

sub send {
        my ($self,$chan,$mess, $orig_mess, $timeout, $func)=@_;
        my $id=$chan->get_mess_id();
        $mess="{".$self->{"name"}." $id}".$mess;
        $self->{$chan}->{$id}->{"timeout"}=time()+$timeout;
        $self->{$chan}->{$id}->{"timeout_func"}=
            sub { &$func($chan,$orig_mess); };
        Reactor::add_event($self->{$chan}->{$id}->{"timeout"},
                    $self->{$chan}->{$id}->{"timeout_func"});
        return $mess;
}

=back

=cut

sub destroy {
        my ($self, $chan) = @_;
        foreach (keys %{$self->{$chan}}) {
            Reactor::remove_event($self->{$chan}->{$_}->{"timeout"},
                                  $self->{$chan}->{$_}->{"timeout_func"});
            delete $self->{$chan}->{$_}; # just to make sure we don't
                                         # leave any circular
                                         # dependencies
        }
        delete $self->{$chan};
}

sub activate {}        # Ack should be invoked directly.
```

```
# Somewhat radical, but as a default action at least it is conservative (no
# timeout callback will be called.)

sub deactivate {
        my $self=shift @_;
        $self->destroy(@_);
}

sub close {
        my $self=shift @_;
        $self->destroy(@_);
}

1;

=head1 CALLBACK PROTOTYPES

=over 4

=item timeout ( CHANNEL, STRING )

The timeout function takes the Channel and the original message.

=head1 AUTHOR

Frederic Dumont <fdumont@info.fundp.ac.be>

=cut
```

Bibliography

[1] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P Anderson Co., Fort Washington, PA, April 1980.

[2] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. Technical report, COAST Laboratory, Purdue University, West Lafayette IN 47907-1398, June 1998.

[3] Peter Cheeseman, Robin Hanson, and John Stutz. Bayesian classification with correlation and inheritance. In *12th International Joint Conference on Artificial Intelligence*, 1991.

[4] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, concepts and design, Edition 2.* Addison-Wesley, isbn0-201-62433-8 edition, 1994.

[5] Mark Crosbie and Gene Spafford. Active defense of a computer system using autonomous agents. Technical Report 95-008, Purdue University, February 1995.

[6] Jan H. P. Eloff and S. H. von Solms, editors. *Information Security – the Next Decade, Proceedings of the 11th International Information Security Conference IFIP SEC'95.* Chapman & Hall, 1995.

[7] K. L. Fox, R. R. Henning, J. H. Reed, and R. P. Simonian. A neural network approach towards intrusion detection. In *Proc. 13th National Computer Security Conference. Information Systems Security. Standards—the Key to the Future*, volume I, pages 124–134, Gaithersburg, MD, 1990. NIST.

[8] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security.* O'Reilly & Associates, 1996.

[9] Naji Habra, Baudouin Le Charlier, Aziz Mounji, and Isabelle Mathieu. ASAX: Software architecture and rule-based language for universal audit trail analysis. In Y. Deswarte, G. Eizenberg, and J.-J. Quisquater, editors, *Proceedings of the Second ESORICS*, Lecture Notes in Computer Science, Toulouse, France, November 1992. Springer-Verlag, Berlin Germany.

[10] Naji Habra, Baudouin Le Charlier, Aziz Mounji, and Isabelle Mathieu. Preliminary report on advanced security audit trail analysis on unix (ASAX also called SAT-X). Technical report, Institut D'Informatique, FUNDP, rue Grangagnage 21, 5000 Namur, Belgium, September 1994.

[11] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. Technical report, University of New Mexico, Department of Computer Science, August 1990.

[12] D. K. Hsiao, D. S. Kerr, and S. E. Madnick. *Computer Security.* Academic Press, New York, 1 edition, 1979.

[13] Irwin and Vazquez. Shadow indications technical analysis. Naval Surface Warfare Center, Dahlgren Division, Code CD2S, 1998.

[14] Deborah G. Johnson and Helen Nissenbaum, editors. *Computers, Ethics & Social Values.* Prentice-Hall, Inc, 1995.

[15] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* Ph.d. thesis, Purdue University, Purdue, IN, August 1995.

[16] Sandeep Kumar and Eugene Spafford. A Taxonomy of Common Computer Security Vulnerabilities based on their Method of Detection. (unpublished), June 1994.

[17] Sandeep Kumar and Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Purdue University, Department of Computer Sciences, March 1994.

[18] Sandeep Kumar and Eugene Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, October 1994.

[19] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. A Real-Time Intrusion Detection Expert System (IDES) – Final Technical Report. Technical report, SRI Computer Science Laboratory, SRI International, Menlo Park, CA, February 1992.

[20] Ludovic Mé. $G^{A}{}_{S}SA_{T}A$, a genetic algorithm as an alternative tool for security audit trails analysis.

[21] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* Doctor of science, Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), September 1997.

[22] Abdelaziz Mounji, Baudouin Le Charlier, Denis Zampunieris, and Naji Habra. Distributed audit trail analysis. Technical Report RP-94-007, Faculté Universitaire Notre-Dame de la Paix, 1994.

[23] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[24] Richard E Overill. How re(pro)active should an ids be?

[25] R Power. Csi roundtable: Experts discuss present and future intrusion detection systems. *Computer Security Journal*, XIV No.1, 1998.

[26] Deborah Russell and G. T. Gangemi Sr. *Computer Security Basics*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991. A clear overview on many different security issues.

[27] Randal L. Schwartz. *Learning Perl*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1993.

[28] Larry Wall, Randal L. Schwartz, Tom Christiansen, and Stephen Potter. *Programming Perl*. Nutshell Handbook. O'Reilly & Associates, 2nd edition, 1996.

Index